Parallel Scientific Computing with Applications in Material Science and Metallurgy

Jon Matteo Church

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy

The University of Leeds
School of Computing
July 22, 2022

Some parts of the work presented in Sections 2.1.3 and 4.3 have been published in the following article:

Jon Matteo Church et al. "High Accuracy Benchmark Problems for Allen-Cahn and Cahn-Hilliard Dynamics". In: *Communications in Computational Physics* 26 (2019), pp. 947–972

The above publication is the joint work of the authors.

Acknowledgements

I which to express my deepest gratitude to both of my supervisors Peter Jimack and Andrew Mullis for their support, advice and for sharing with me their knowledge and expertise over the last years. I'd like to thank all the friend that have shared with me this process. In particular I'd like to thank Giulia and Mark with whom I shared many discussion and lunches.
I am deeply thankful to Maria Teresa for the patience and support she has provided, especially in the final stages of writing during the pandemic. Finally I'd like to thank the examiners for reading my manuscript, and I'd like to acknowledge the School of Computing for funding my research.

Abstract

A new software tool for the solution of complex time-dependent systems of partial differential equations is presented. High levels of parallelization are achieved in a framework that allows the application developer to implement ad-hoc solvers for computationally challenging problems on a higher abstraction level without the need to understand in the low-level parallel implementation. Moreover, thanks to this new implementation, advanced numerical methods, such as mesh adaptivity, implicit time stepping, and multigrid methods can be employed with ease. Here the implementation of this new tool is presented and validated against simple elliptic and more complex phase-field models. Its parallel performance is then assessed.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A deep understanding of the phenomena that occur in an alloy during its solidification process is of fundamental importance in metallurgy applications such as the casting industry. Castings are made by pouring melted metals into a mold allowing the production of items whose shape could be impossible to obtain with other techniques. During the transformation from the liquid to the solid state in the mold, many small crystals are formed in the metal affecting the final properties of castings [13].

The primary motivation for this research is to extend an existing open source software tool to allow it to solve complex time-dependent systems of PDEs, including models for phase change, to allow investigation of phenomena such as crystal dendrite formation using advanced numerical techniques.

This work adds new functionalities to an existing parallel computational framework and implements a general abstract interface that allows the solution not only of phase-field models, but of a wide family of problems that can be modelled by systems of time-dependent partial differential equations.

1.1 Litterature Review

In recent literature, phase field models represent a standard approach in modelling systems where two or more phases of the same component coexist [16, 27, 48, 49, 56, 65]. The mathematical idea behind these models, in the case of two phases, is to introduce a continuous auxiliary function φ defined over the spatial domain, $\Omega \subset \mathbb{R}^N$, that assumes fixed values in each bulk phase and that smoothly varies between them over a thin region where the actual phase transition occurs. Note that an alternative to the phase-field approach is the level-set method (e.g. [89]), which represents the interface between two phases as the zero level set of an auxiliary function: however we do not consider the level set approach in this thesis

Numerical simulations, both in 2D and 3D, have been performed by other authors using phase field models: Narski and Picasso [70], for example, used phase field models with an adaptive isotropic mesh, Tonhard and Amberg [91] and Lin et al. [60], instead, used structured isotropic adaptive mesh and the finite difference method, while Sarkis et al. [87] used the finite element method over an anisotropic unstructured adaptive mesh, and Guo and Xiong [37] implemented a finite volume scheme with structured adaptive mesh refinement. These works demonstrate the need of adaptive mesh refinement techniques while solving the phase field model. However, all of them made use of explicit time stepping, while it has been shown by Rosam et al. [79, 80] that an implicit scheme is preferable for some families of phase-field models once a certain level of refinement is reached. The same authors [7, 34] have shown that the system of the non-linear equations resulting from the application of a finite difference method can be solved efficiently with non-linear multi-grid techniques. An analogue solver for block-structured grids has been used also by Wise et al.[100] for the efficient solution of a similar set of equations but with applications within biomedical engineering.

The trade-off between accuracy and computational time often leads researchers to carry out simulations in two dimensions or with model parameters that are far from resembling configurations of physical interest. Without such simplification, numerical simulations would simply take too long and require too much memory.

To overcome such limitation, an highly scalable solver is necessary in order to achieve results of physical interest within reasonable time, and overcome the sub-optimal scalability properties of the implementation in [7].

Given the scale of the problems we aim to solve, this research focuses on the block-structured adaptive mesh refinement method (SAMR), which has been extensively used over the last decade in computational high-level frameworks. Few SAMR frameworks, however, are open source, actively developed, and have been used also by users out of the organizations developing them.

At the beginning of this research, BoxLib [106], Chombo [21], and SAMRAI [44] where the popular general frameworks that provided all the functionality for solving different systems of hyperbolic, parabolic and elliptic equations. PARAMESH [63], was an even more general library the only implements the mesh management, and therefore is completely independent on the equations being solved. A more exhaustive list of the software available then that implement SAMR, as well as other adaptive mesh refinement (AMR) techniques can be found at [38].

BoxLib is a Fortran/C++ software framework that provides extensive support for both explicit and implicit grid-based and particle fields on hierarchical adaptive meshes. It includes multigrid solvers for both cell-based and node-based data. Parallelism is mainly achieved representing the grid as the union of blocks at each level and organizing data in arrays on each single block. Operations spanning across these blocks are distributed among the computational nodes, with arrays at each level of refinement being distributed independently. Each node stores all meta-data needed to describe the whole geometry and processor assignments. To optimize communications patterns, this meta-data must be used to dynamically change communication patterns and share data amongst nodes by each specific algorithm. The scaling behavior of BoxLib is therefore strongly dependent on the specific algorithm.

Chombo is also a mixed-language software framework that uses C++ for high-level abstractions and Fortran for calculations on regular patches. It supports a wide variety of applications that use AMR by means of a common software framework as well as built-in diagnostic and performance tools. Its approach to parallelism is similar to AMReX's, with the main difference being that meta-data is replicated across all MPI ranks to reduce communication. Moreover, data in Chombo is templated on data type and data centering and has a hierarchy: from grid data to global. Moreover, Chombo provides support to embedded boundary algorithms [22].

Enzo's SAMR implements rectangular blocks of arbitrary size and aspect ratios and adaptive time-stepping. Each refinement level uses its own timestep. Also Enzo uses C++ for the data, comunications and task managment and Fortran-90 for its local solvers. Non-local solvers however require substantial amounts of communication. Two level of concurrency are supported: block-level and deeper level. Its development is very active since the project is fully open source and community-driven. Much effort was given at that time to port solver to graphics processing units (GPUs), and using the hybrid-parallelism.

FLASH is a component base software that combines two frameworks, an Eulerian discretized mesh and a Lagrangian approach [76]. FLASH support explicit stencil-type solvers using PARAMESH and Chombo. Only explicit solvers are completely agnostic of the underlying mesh, elliptic and parabolic solvers interact explicitly with the mesh using the PARAMESH interface.

For this research, the the Uintah Computational Framework (UCF) has been chosen, because of its approach to parallelism and for it being, at that time, perhaps ahead of others in terms of programming abstractions. It consists of software components and libraries that can execute efficiently multi-physics simulations by identifying its simulation stages and automatically analyzing their dependencies and communication patterns. Similarly to Charm++ [46], Uintah uses task-graphs, but also implements a data warehouse which manages data transfers, and which ensures that the user's code is independent of the communications layer. Uintah's makes it possible to improve scalability through adaptive self-tuning, making its scaling behaviour independent on the specific algorithm.

Uintah's user base has continuously expanded since 2008, and kept improving its scalability properties. Uintah-based simulations of next generation coal boilers have been successfully performed on different architectures, including heterogeneous architectures and have scaled up to 96K, 262K, and 512K cores on the NSF Stampede, DOE Titan, and DOE Mira, respectively [6, 68].

During this research the other framework kept improving as well, but other frameworks had to undergo relatively greater changes than Uintah. While Uintah, thanks to its component based organization simply tested new technologies such as Kokkos and Hedgehog [41], Enzo has been completely rewritten from scratch (Enzo-E), Flash underwent an architectural refactoring

(Flash-X) [26], and BoxLib has been replaced by AMReX [105].

Given the many changes in the SAMR software panorama during the years of this research, Uintah has proven to be a solid choice as development framework, allowing my work to progress without being affected by its continuous development.

1.2 Contribution to Uintah

The Uintah Computational Framework is a continuously developing SAMR software which is organized in well separated components. When I started this research work, although all the components required to perform time-dependent partial differential equations where available, not all of them could work well together. In particular, the current implicit solver interface component wasn't designed to work with multi-level structured grids. An earlier and unmaintained and AMR implicit solver implementation was present in the code base, but it had never worked properly. It was developed for a specific application and the main problem was related with handling the interfaces between fine and coarse levels without introducing numerical instabilities. For this application an ad-hoc smoothing operator was sufficient to control this issue.

The main challenge and contribution of my work to Uintah is the development of a fully functional and generic implicit solver component that acts as a bridge between the AMR structures of Uintah and the semi-structured solver interface of HYPRE. HYPRE has been the natural choice for the linear algebra library, since also the Uintah's current implicit solver component for single level grids did already use it.

The second contribution to Uintah is the introduction of a new abstract family of templated classes that act as viewers of the SAMR data structures in Uintah's data warehouse. This provides the final user a higher level of abstraction to implement finite differences/volume discrete models. In fact, all previous applications of Uintah were specialized for solving specific model equations and each for each one specific kernels were implemented at the patch level to assemble matrices and to progress the solution from one timestep to the next.

This research, on the contrary, targets multiple applications and different model equations. For this reason the introduction of such an higher level interface has been crucial to allow a much more straightforward implementation of kernels at the patch level. This same high level interface also hides the complexity of boundary and fine/coarse interface conditions so that the same kernel can be used for any patch on any level and any combination of conditions.

Thanks to this interface, it is possible to configure boundary conditions and fine/coarse interface strategies in the problem specification file, which is loaded at run time, giving much greater flexibility to the final user of Uintah.

1.3 Thesis Structure

This thesis is structured in nine chapters in addition to this introduction.

Chapter 2 is composed of three sections which provide an overview of the physical models, the software tools and the numerical techniques on which this research work is based. In §2.1 some phase field models for describing the solidification process of pure metals are introduced; in particular, the Allen–Cahn and Cahn–Hillard equations are discussed in §2.1.3. Four problems of these types have been identified as representative of the challenges that can arise during phase field simulations and will then be used to benchmark the software implementation produced by this research. In §2.1.1, the isotropic and the anisotropic models which describe the solidification problem of pure metals is introduced.

In §2.2 the Uintah Computational Framework is presented, focusing on how high degrees of parallelism are achieved by the framework. In §2.2.2, the different available schedulers are presented, then, in §2.2.3, the organization in levels and patches of adaptive meshes is described. How data is stored and how it is shared between processes within distribute memory architectures is discussed in §2.2.4. Ultimately, in §2.2.4, the mesh regridder within the UCF is presented.

A brief introduction of the numerical techniques implemented in this research work is provided in §2.3. In §2.3.1 the simpler heat diffusion equation is introduced as a model problem to showcase such techniques. The use of finite differences to discretize spatial differential operators is described in §2.3.2, while the time stepping schemes are discussed in section §2.3.2. The last section of the chapter, §2.3.3 is dedicated to a brief presentation of the numerical techniques used to solve the linear systems that arise when non explicit time schemes are implemented.

Chapter 3 presents the overall research approach and how the software components developed and evaluated in Chapters 4–8 come together. The same chapter also articulates what the challenges are in working with a large software package such as Uintah.

In Chapter 4 a short overview of the initial stages of the software development of this research project is provided. The very first implementation in C++ of an explicit pure metal solver without mesh adaptivity is described in §4.1. The following section, §4.2 describes how the same problem is implemented within the UCF using the same numerical techniques. Parallelism is achieved by this implementation but the applications of this solver are very limited due to the restrictions imposed by the explicit time scheme and the uniformity of the computational grid. In the last section of the chapter, §4.3, a solver to the simpler heat diffusion problem is implemented. This implementation provides a base implementation on top of which more advanced numerical techniques are built and tested. The implementation of boundary condition mesh adaptivity for this solver showed that a different and more abstract approach was needed to provide a tool for the implementation of phase field components within the UCF. Without this abstract approach, error prone and heavily duplicated code would have been necessary each time an additional phase model model was added to the framework.

Chapter 5 describes the details of the implementation which allows this more abstract approach. In particular, the implementation of the view framework which provides for a general interface to spatial finite differences operators is described for periodic geometries. The general view framework is then extended in Chapter 6 and Chapter 7 to support bounded computational domains and adaptive grid refinement respectively. In Chapter 8, details on how linear solvers are interfaced by Uintah are given. In particular, how to use the existing driver to the HYPRE library [29] is described in §8.2.1. An additional driver which supports structured grids (a general class of grids that includes adaptive grids) is also implemented §8.2.2. Each of these three chapters is structured in three sections: the first one is an analysis of the requirements of the implementation at that stage of development, then details on how that requirements are matched by the implementation are provided, and, ultimately, appropriate test cases are selected to validate the new implementation.

The next Chapter 9 analyses the behaviour of the implementations developed during this research in terms of parallel performance.

The last Chapter 10 summarizes the achievements of this research and provides for ideas on how this work can be extended and further developed.

Chapter 2

Background

2.1 Phase-Field Models

In recent literature, phase field models represent a standard approach in modelling systems where two or more phases of the same component coexist [56, 27, 16, 48, 49]. The mathematical idea behind these models, in the case of two phases, is to introduce a continuous auxiliary function ψ defined over the spatial domain, $\Omega \subset \mathbb{R}^N$, that assumes fixed values in each bulk phase and that varies smoothly between them over a thin region where the actual phase transition occurs.

The evolution of the interface is modelled by non-linear partial differential equations making it unnecessary to track explicitly the solid phase front.

2.1.1 Pure Metal Isotropic solidification

The simplest phase field model that has been considered describes the process of isotropic solidification of a pure undercooled metal [48]:

$$\begin{aligned} \tau \partial_t \psi &= W^2 \Delta\psi + \partial_\psi f(\psi, u; \lambda) \\ \partial_t u &= \alpha \Delta u + \tfrac{1}{2} \partial_t h(\psi) \ , \end{aligned} \tag{2.1}$$

where the unknowns are: the phase field, $\psi : \Omega \to [-1, 1]$, which varies from -1 in the liquid to 1 in the solid; and the dimensionless temperature field, $u : \Omega \to \mathbb{R}$, defined by

$$u := \tfrac{T - T_M}{L/c_p} \ ,$$

where T is the temperature field, T_M is the melting temperature, L is the latent heat of melting, and c_p is the specific heat at constant pressure. The parameters used in this model are: the characteristic time of attachment of atoms at the interface, τ, whose typical value is about

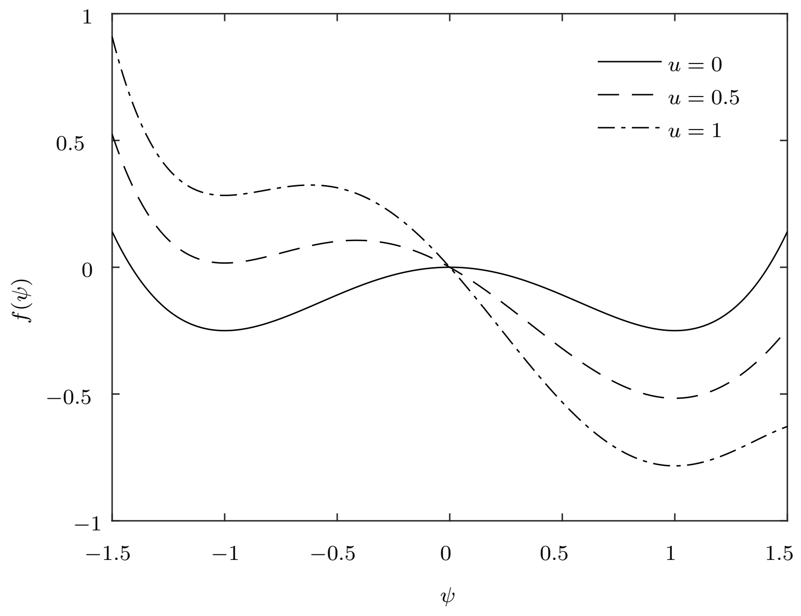


Figure 2.1: Graph of the potential, f, for three different values of non dimensional temperature, u, given $\lambda = 1$.

10^{-13} s; the interface thickness, W, in the order of the Angstrom; the thermal diffusivity, α; and the dimensionless coupling parameter, λ, that controls the strength of the the coupling between the two fields. The remaining terms in (2.1) represent a double-well potential, $f(\cdot, \cdot; \lambda) : [-1, 1] \times \mathbb{R} \to \mathbb{R}$, and a latent heat source, $h : [-1, 1] \to \mathbb{R}$.

This model is a combination of both dimensional and non-dimensional quantities; in fact, space and time domains are dimensional as well as the parameters τ, W, and α, while the temperature and phase fields are dimensionless as is the coupling parameter λ.

The function f can be chosen to be defined by

$$f(\psi, u; \lambda) := \left[\frac{\psi^4}{4} - \frac{\psi^2}{2}\right] - \lambda u \left[\frac{\psi^5}{5} - \frac{2\psi^3}{3} + \psi\right] , \tag{2.2}$$

which is double-well potential where the barrier between the solid and the liquid phase varies with the temperature. See Figure 2.1 for examples of its graph at different values of u.

Eqns. 2.1 can be derived from the variational form of a single Lyapunov functional as in Langer [58], which has been later interpreted as the total entropy of the system in a given volume [74, 94].

2.1.2 Pure Metal Anisotropic Solidification

The isotropic model (2.1), as others [56, 99, 93, 65, 48], can be extended to describe the process of anisotropic solidification by introducing a functional dependence of τ and W upon the normal

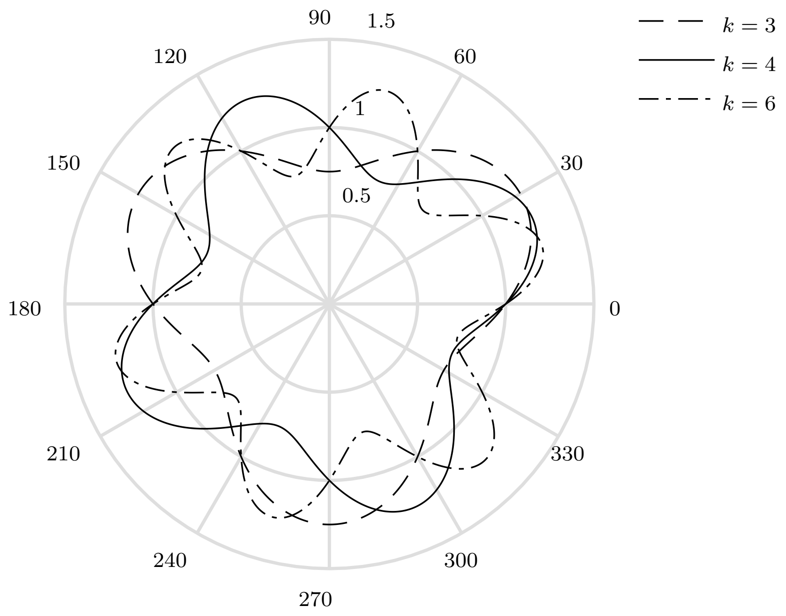


Figure 2.2: Plot of the anisotropy function, A, for three different values of mode number, k, given $\varepsilon = \frac{1}{4}$.

direction to the interface,

$$\boldsymbol{n} := \tfrac{\nabla\psi}{|\nabla\psi|} \ ,$$

which is is achieved by setting $\tau \equiv \tau(\boldsymbol{n}) := \tau_0 A(\boldsymbol{n})$ and $W \equiv W(\boldsymbol{n}) := W_0 A(\boldsymbol{n})$.

In the derivation of the model equations, as for the isotropic case, the variational derivative of the total entropy must be computed which, due to the dependency W and τ upon $\boldsymbol{n}$, leads to a more complex system of equations:

$$\begin{aligned} \tau_0 A \partial_t \psi &= W_0^2 \nabla\cdot(A^2 \nabla\psi) + W_0^2 \nabla\cdot\left[|\nabla\psi|^2 A\, \partial_{\nabla\psi} A\right] + \partial_\psi f(\psi, u; \lambda) \\ \partial_t u &= \alpha\Delta u + \tfrac{1}{2}\partial_t h(\psi) \ . \end{aligned} \tag{2.3}$$

In two dimensions it is common practice to define the azimuth angle, φ, between $\boldsymbol{n}$ and the axis of abscissae ($y = 0$),

$$\varphi := \arctan \tfrac{\partial_y \psi}{\partial_x \psi} \ ,$$

and setting the anisotropy function to

$$A(\varphi; \varepsilon, k) := 1 + \varepsilon \cos(k\varphi) \ , \tag{2.4}$$

which is parametrized by the anisotropy strength, ε, and the mode number, k.

With this choice of coordinates for the anisotropy function it is possible to rewrite (2.3) as follows:

$$\begin{aligned}\tau_0 A\partial_t\psi &= W_0^2\nabla\cdot(A^2\nabla\psi) - W_0^2\partial_x[AA'\partial_y\psi] + W_0^2\partial_y[AA'\partial_x\psi] + \partial_\psi f(\psi,u;\lambda)\\ \partial_t u &= \alpha\Delta u + \tfrac{1}{2}\partial_t h(\psi)\ .\end{aligned}\tag{2.5}$$

The shape of the anisotropy function governs the direction along which the dendrite growth is favoured. In Figure 2.2 a polar plot of A is shown to illustrate the shapes it assumes for several values of the mode number.

The choice $k = 4$, for example, corresponds to modelling a crystal whose growth is favoured along the main Cartesian axes. In the literature other anisotropy functions are used to take into account, for example, asymmetry between extreme values of A, as in [107], or faceted crystals [25, 69].

In three dimensions, anisotropy function is commonly chosen to favour the growth of crystal branches along the Cartesian axes [49, 48, 15, 59, 7]

$$A(\varphi,\vartheta;\varepsilon) := 1 + \varepsilon\cos(4\varphi) - 2\varepsilon\sin^2(2\vartheta)\sin^4(\varphi)\ ,\tag{2.6}$$

where φ, and ϑ are respectively the azimuthal and zenith angle of the normal versor to the interface in spherical coordinates. In Figure 2.3 a spherical plot of A is shown.

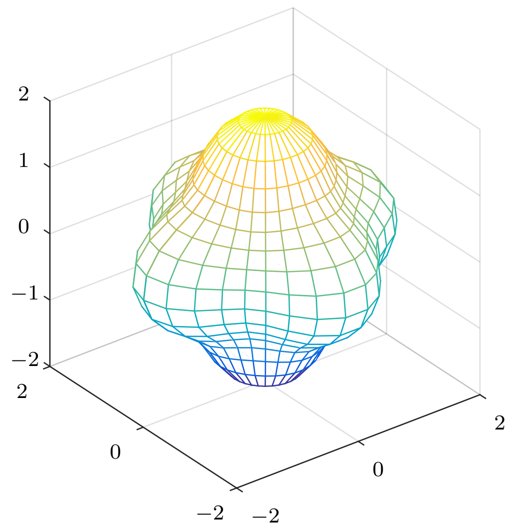


Figure 2.3: Plot of the three dimensional anisotropy function, A, for four fold symmetry, given $\varepsilon = 1$.

Formulations such as (2.4) and (2.6), where anisotropy is defined as a function of angular coordinates, allow to express straightforwardly their parametric dependence on the mode number but, at the same time, introduce some singularities where such angles are not well defined.

By means of the trigonometric identities for multiple angles, it is possible to expand those formulation as linear combinations of powers of sines and cosines of the angular coordinates which can than be identified with the components of the normal vector, $\boldsymbol{n}$.

The resulting expression can than be reformulated within the spherical or cubic harmonics formalism, depending on the dimension of the model [103]. In general, all formulations of the anisotropy function can be expressed as linear combinations of spherical harmonics, in two dimensions, or cubic harmonics in three dimensions [30, 75].

In this regard, the following expressions for the anisotropy function are adopted in this work for the two and three dimensions cases respectively:

$$A_{2\mathrm{D}}(\boldsymbol{n}) := 1 - 3\varepsilon + 4\varepsilon(n_x^4 + n_y^4)\ , \qquad A_{3\mathrm{D}}(\boldsymbol{n}) := 1 - 3\varepsilon + 4\varepsilon(n_x^4 + n_y^4 + n_z^4)\ , \tag{2.7}$$

which correspond, the first, to (2.4) for $k = 4$ and, the latter, to (2.6). Since the normal vector to the interface, $\boldsymbol{n}$, is well defined only for $|\nabla\psi| \neq 0$ and we want to extend the functions above to the whole computational domain, we can embed our N-dimension problem into $\mathbb{R}^{N+1}$. In this way, where $|\nabla_N\psi| = 0$, the normal vector to the interface is the unitary vector orthogonal to the original N-dimension subspace and the anisotropy function assumes the constant value $1 + \varepsilon$.

Due to the artificial definition of ψ, these models assume a quantitative meaning only in the *sharp-interface limit*, when the interface thickness between the liquid and the solid phases is negligible. In this limit, Eqns. 2.3 reduce to the standard free-boundary formulation [48]:

$$\begin{aligned} \partial_t u &= \alpha\Delta u \\ v &= \alpha\left[\partial_{\boldsymbol{n}}^{-} u - \partial_{\boldsymbol{n}}^{+} u\right] \\ u_i &= -\kappa d_0 - \beta v\ , \end{aligned} \tag{2.8}$$

where v, u_i, and κ are the normal velocity, the temperature, and the principal curvature of the interface, respectively. Parameters d_0 and β are the capillary length and the kinetic coefficient and are linked to those in model (2.3) by the following [49] relations:

$$d_0 = a_1 \tfrac{W}{\lambda}\ , \qquad \beta = a_1\left[\tfrac{\tau}{\lambda W} - a_2\tfrac{W}{\alpha}\right]\ , \tag{2.9}$$

where $a_1, a_2 \in \mathcal{O}(1)$ are positive constants depending on the choice of f and h.

The usual initial condition for both the isotropic model (2.1) and the anisotropic one (2.5) describes a solid seed surrounded by an undercooled liquid phase:

$$\begin{aligned} \psi(\mathbf{x}) &= -\tanh(\gamma_\psi[|\mathbf{x} - \mathbf{x}_0|^2 - R_0^2]) \\ u(\mathbf{x}, 0) &= -\tfrac{\Delta}{2}[1 - \tanh(\gamma_u[|\mathbf{x} - \mathbf{x}_0|^2 - R_0^2])]\ , \end{aligned} \tag{2.10}$$

where the γ's control the initial interface width of the phase and the temperature fields, Δ is the initial undercooling and $R - 0$ and $-\mathbf{x}_0$ are the initial seed radius and centre.

More complex models are available in literature for describing crystal dendrite formation process in solidifying metals to take into account binary or higher component alloys [77], heat transfer phenomena[48] and chemical reactions [47].

These have been considered in this research but it has been decided to focus on the implementation of the computational framework, therefore only the simplest models have been implemented, leaving the development of these more complex models to future research.

2.1.3 Benchmark Models[1]

Phase field models are largely used in materials science and there is a large literature of numerical methods for them. The prototype models are the Allen-Cahn and Cahn-Hilliard equations and, as such, these models can be used to benchmark the software implementations of this research work. Four benchmark problems for these equations have been identified together with a wide group of researchers (Zhenlin Guo, University of California, Irvine; Keith Promislow, Michigan State University; Brian Wetton, University of British Columbia; Steven Wise, University of Tennessee Knoxwille; Fengwei Yang, University of Sussex) as representative of numerically challenging problems. Several numerical techniques have been compared together with the aim of computing benchmark values on which most methods could agree to be used as references by researcher in the broad community to assess the reliability of their phase field implementations. A deeper discussion of the techniques used and of the outcome of this joint research is available in [20]. The goal of this paper is to provide the scientific community with a reliable reference point for assessing the accuracy and reliability of future software for this important class of problem.

Allen–Cahn Equation

Benchmark I: 2D Allen Cahn The first benchmark is for the Allen-Cahn equation [1]:

$$\partial_t u = \varepsilon^2 \Delta u - W'(u), \qquad \text{in } \Omega,\ t \geq 0, \tag{2.11}$$

where $W(u) := \frac{1}{4}(u^2 - 1)^2$. It describes the evolution of crystal grains of the same material during annealing. It can also be called a Ginzberg-Landau equation. The chosen benchmark problem is a simple 2D problem in a doubly periodic domain

$$u_{|x_i=0} = u_{|x_i=2\pi}, \qquad \text{on } I,\ t > 0,\ i = 1, 2, \tag{2.12}$$

[1]Some parts of the work presented in this section have been published in [20].

with initial conditions

$$u(\boldsymbol{x},0) = \tanh\left(\frac{|\boldsymbol{x}-\boldsymbol{c}|-2}{\varepsilon\sqrt{2}}\right), \qquad \boldsymbol{x} \in \Omega, \tag{2.13}$$

with $I = [0, 2\pi]$, $\Omega = I^2$, $\boldsymbol{c} = (\pi, \pi)^t$, and $\varepsilon = 0.2,\ 0.1,\ 0.05$.

The benchmark is the time T at which the value at the domain centre $\boldsymbol{c}$ changes from negative to positive.

The free energy of a system described by the Allen-Cahn model can be expressed as the following integral:

$$\mathscr{E} = \int_\Omega \tfrac{1}{2}\varepsilon^2 |\nabla u|^2 + W(u)\, dx. \tag{2.14}$$

Some snapshots of the dynamics are shown in Figure 2.4. A video of the dynamics is also available [96].

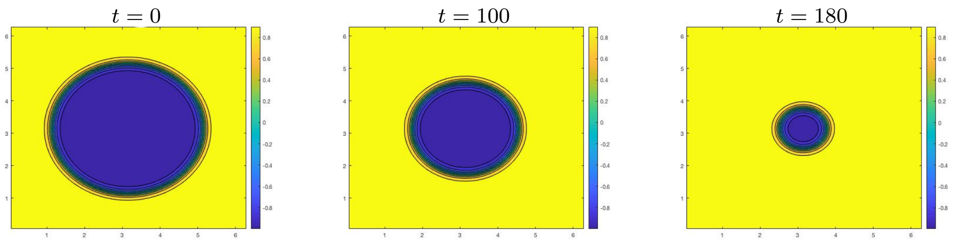


Figure 2.4: Benchmark I: Allen Cahn dynamics with $\varepsilon = 0.1$.

Cahn–Hillard Equation

The Cahn-Hilliard (CH) [11] equation that describes phase separation of a binary alloy during annealing. The problem is described by a scalar function u of space $\mathbf{x}$ and time t that takes values $u = +1$ in one phase and $u = -1$ in the other.

$$\partial_t u = -\varepsilon^2 \Delta^2 u + \Delta[W'(u)], \qquad \text{in } \Omega,\ t \geq 0 \tag{2.15}$$

where $W(u) = \frac{1}{4}(u^2 - 1)^2$ and Δ is the Laplacian operator. The parameter ε in the model is a length scale – the width of the layers between the regions of different phases.

The study of equilibrium of the Cahn-Hilliard equation is equivalent to the study of the minimizers of the Cahn-Hilliard free energy

$$\mathscr{E}(u) := \int\limits_\Omega \tfrac{1}{2}\varepsilon|\nabla u^2| + \varepsilon^{-1} W(u)\, dx\,. \tag{2.16}$$

Note that the benchmark problems focus on pure materials science applications rather than the use of Cahn-Hilliard equations to track interfaces in so-called *diffuse interface methods* [104, 8] in which the CH dynamics are coupled to other physics.

Benchmark II: 2D Cahn Hilliard seven circles The second benchmark is for the 2D Cahn Hilliard dynamics (2.15), again in the doubly periodic domain $\Omega = I^2$

$$u_{|x_i=0} = u_{|x_i=2\pi}, \qquad \text{on } I,\ t > 0,\ i = 1, 2, \tag{2.17}$$

with $I = [0, 2\pi]$.

Initial conditions are seven circles with different centres and radii dressed with a smooth profile:

$$u(\boldsymbol{x}, 0) = -1 + \sum_{n=1}^{7} \psi\left(\frac{|\boldsymbol{x}-\boldsymbol{c}_n| - r_n}{\varepsilon}\right), \qquad \boldsymbol{x} \in \Omega, \tag{2.18}$$

where

$$\psi(s) := \begin{cases} 2\exp(-s^{-2}) & \text{if } s < 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\begin{aligned}
\boldsymbol{c}_1 &= (\tfrac{\pi}{2}, \tfrac{\pi}{2})^t & r_1 &= \tfrac{\pi}{5} \\
\boldsymbol{c}_2 &= (\tfrac{\pi}{4}, \tfrac{3\pi}{4})^t & r_2 &= \tfrac{2\pi}{15} \\
\boldsymbol{c}_3 &= (\tfrac{\pi}{2}, \tfrac{5\pi}{4})^t & r_3 &= \tfrac{2\pi}{15} \\
\boldsymbol{c}_4 &= (\pi, \tfrac{\pi}{4})^t & r_4 &= \tfrac{\pi}{10} \\
\boldsymbol{c}_5 &= (\tfrac{3\pi}{2}, \tfrac{\pi}{4})^t & r_4 &= \tfrac{\pi}{10} \\
\boldsymbol{c}_6 &= (\pi, \pi)^t & r_6 &= \tfrac{\pi}{4} \\
\boldsymbol{c}_7 &= (\tfrac{3\pi}{2}, \tfrac{3\pi}{2})^t & r_7 &= \tfrac{\pi}{4}
\end{aligned}$$

Computations are done with $\varepsilon = 0.1$, 0.05, and 0.025. The benchmarks are the times T_1 and T_2 at which the value at the points $(\pi/2, \pi/2)$ and $(3\pi/2, 3\pi/2)$ change from positive to negative. Some snapshots of the dynamics are shown in Figure 2.5. A video of the dynamics is also available [98].

Benchmark III: 1D Cahn Hilliard This problem was originally proposed in [17]. It is set in the periodic domain $x \in \Omega = [0, 2\pi]$

$$u_{|x=0} = u_{|x=2\pi}, \qquad t > 0, \tag{2.19}$$

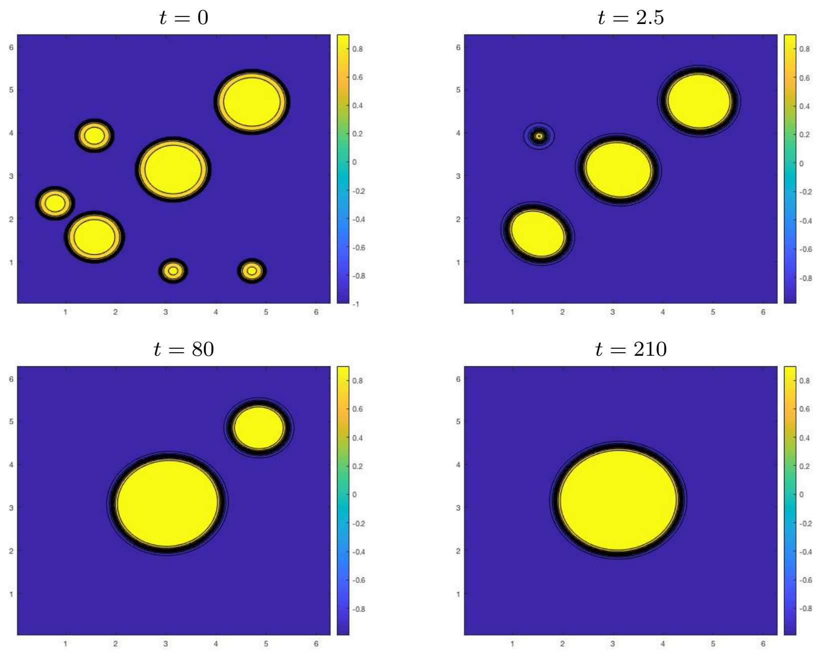


Figure 2.5: Benchmark II: Allen Cahn dynamics with $\varepsilon = 0.05$.

with $\varepsilon = 0.18$ and initial data

$$u(x,0) = \cos(2x) + \tfrac{1}{100}\exp(\cos(x + \tfrac{1}{10})), \qquad x \in \Omega. \tag{2.20}$$

Over a short time, the solution tends to two intervals each of values close to ± 1 with interfaces of width ε between them. The second term on the right is a small perturbation so that these intervals are not symmetric. At very large times, the intervals will slowly (exponentially slow in ε) evolve and merge [73, 78] as shown in Figure 2.6. A video of the dynamics is also available [95]. The final state with two transition layers is steady. The benchmark is the time T at which the midpoint value $u(\pi, t)$ changes from positive to negative. This ripening event happens at a very fast time scale after the long, slow transient. It is the wide range in time scales of the dynamics that makes this a challenging computation.

Benchmark IV: 2D Cahn Hillard Energy Decay This is a modified version of the benchmark proposed in [45]. When scaled, their formulation of Cahn Hilliard is equivalent to (2.15) in the periodic domain $\Omega = I^2$

$$u_{|x_i=0} = u_{|x_i=2\pi}, \qquad \text{on } I,\ t > 0,\ i = 1, 2, \tag{2.21}$$

with $I = [0, 2\pi]$ and $\varepsilon = \frac{\pi\sqrt{10}}{500}$. Their proposed initial conditions have discontinuities at the periodic boundary conditions which implies infinite initial energy, and the early dynamics are

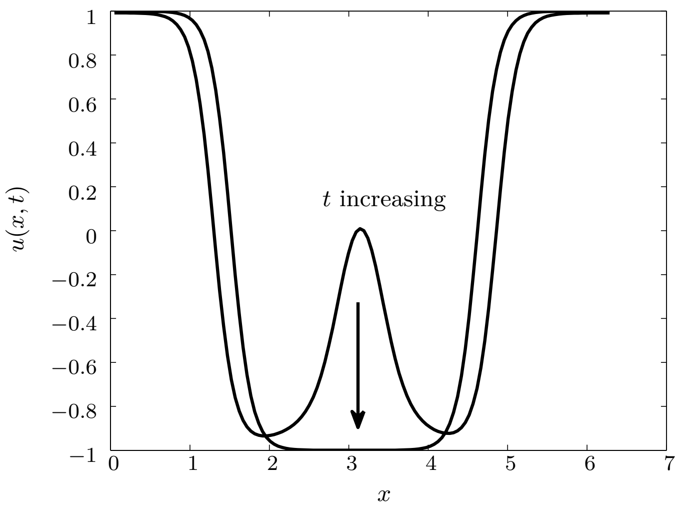


Figure 2.6: This figure corresponds to the solution of the 1D Cahn-Hilliard equation (2.15) with initial conditions (2.20) for $\varepsilon = 0.18$ near the benchmark time.

dominated by the smoothing of these discontinuities. We replace their initial conditions with smooth, periodic ones that give roughly the same energy decay that will be the target of the benchmark:

$$u(\boldsymbol{x},0) = \tfrac{\cos(3x_0)\cos(4x_1)+\cos^2(4x_0)\cos^2(3x_1)+\cos(x_0-5x_1)\cos(2x_0-x_1)}{20}, \qquad \boldsymbol{x} \in \Omega. \tag{2.22}$$

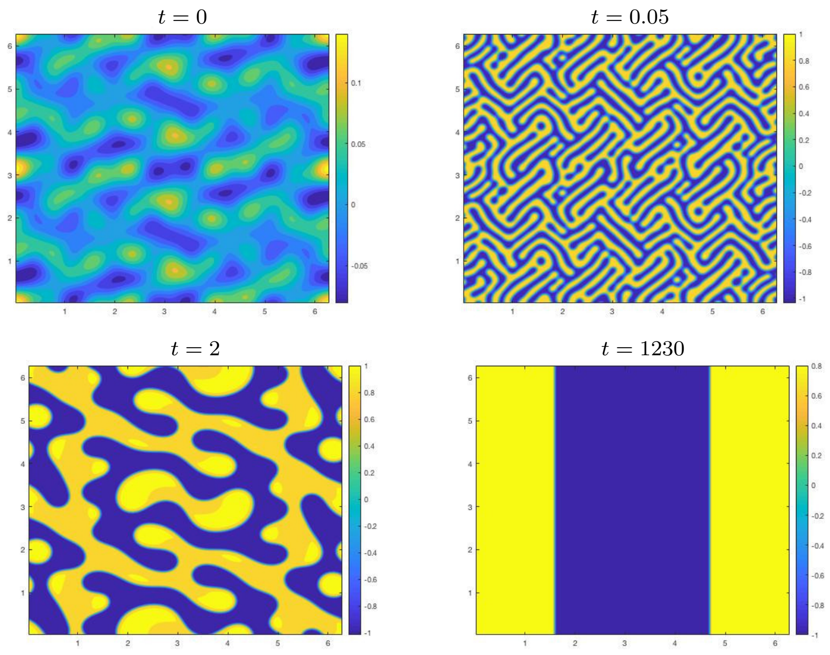
Some snapshots of the dynamics are shown in Figure 2.7 and a video of the dynamics is available [97]. The plot of $\ln\mathscr{E}$ versus $\ln t$, where $\mathscr{E}$ is the energy (2.16) and natural logarithms are used, is shown in Figure 2.8. It is the L_1 error to this function that is the benchmark. Specifically, the differences $\mathscr{D}_1$ and $\mathscr{D}_2$ between the exact $\mathscr{E}_*(t)$ and computed $\mathscr{E}_c(t)$ is given by the benchmarks

$$\mathscr{D}_1 = \int_{-5}^{7} |\ln\mathscr{E}_*(\vartheta) - \ln\mathscr{E}_c(\vartheta)| d\vartheta \tag{2.23}$$

$$\mathscr{D}_2 = \int_{-5}^{2} |\ln\mathscr{E}_*(\vartheta) - \ln\mathscr{E}_c(\vartheta)| d\vartheta \tag{2.24}$$

where $\vartheta = \ln t$. Pointwise values of an accurate approximation of $\mathscr{E}_*(t)$ can be found online [101]. For the accuracy reported in our computations, approximating the integrals in $\mathscr{D}_{1,2}$ with Trapezoidal rule and 1,000 equally spaced points in the interval, using linear interpolation of the computed $\mathscr{E}$ values, is sufficient.

These are proposed because often in applications the exact details of the computational results are not important, but the trend of the evolution of length scales is a key feature [5].



ch

Figure 2.7: Benchmark IV: Cahn Hillard Energy Decay.

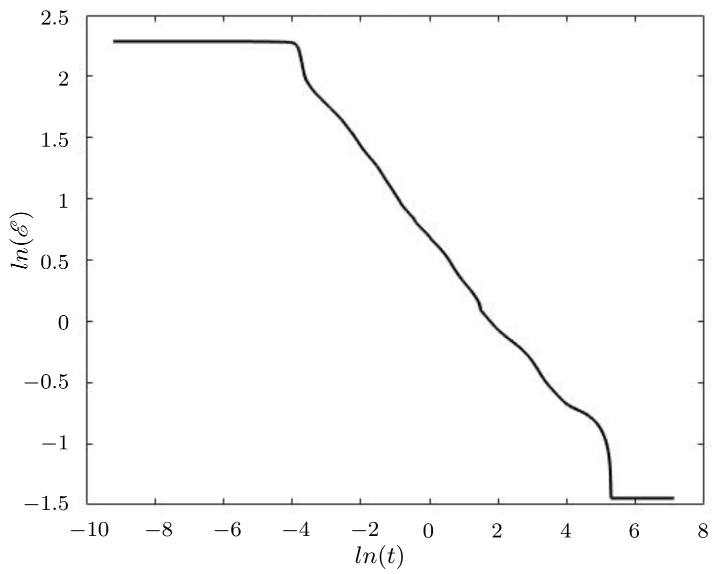


Figure 2.8: This figure shows the energy decay profile for the benchmark IV: 2D Cahn Hilliard problem.

The difference $\mathscr{D}_1$ measures the difference over the full dynamics, while $\mathscr{D}_2$ covers only the first part of the dynamics and omits the fine details of the final transition to steady state.

2.2 Uintah

2.2.1 Introduction

The Uintah Computational Framework (UCF), is a collection of software developed for running on parallel systems with the purpose of solving partial differential equations (PDEs) on structured grids implementing adaptive mesh refinement (AMR). The software is structured in self-contained modules (components) and libraries following the specifications of the Common Component Architecture (CCA).

UCF started as a project funded by the U.S. Department of Energy and developed by the Center for Simulation of Accidental Fires and Explosions (C-SAFE) at the University of Utah to be, then, released to to the broader scientific and engineering community in 2009.

Its component based architecture allows the developer to integrate together different simulation components leaving the framework itself to identify their dependencies and interfaces. Tasks are represented in Uintah by abstract graphs describing the computation scheme and communication [72, 71] which are interpreted and analysed by a specific UCF component that

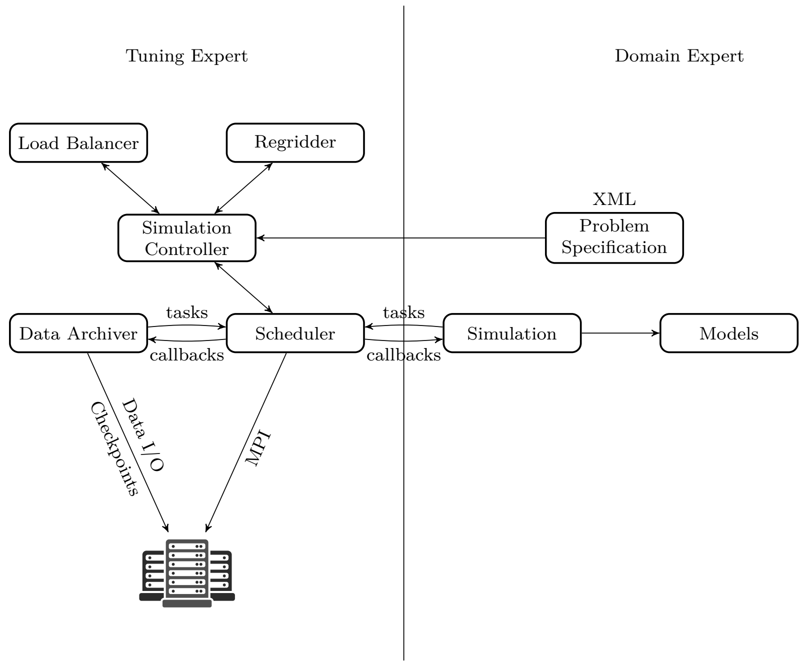


Figure 2.9: Uintah Framework separation between parallel components and simulation components which belong to different expertise fields (edited from [62]).

is capable to automatically identify the best workloads and communication patterns and achieve optimal global resource utilization during parallel multi-physics simulations. Moreover, its modular approach makes it possible for the UCF to maintain overall scalability even when different components are adopted jointly.

Thanks to the aforementioned Uintah's ability to analyse the computation scheme, the developer can take advantage of automatic load balancing, parallel input/output, breakpoints and suspend/restart procedures.

Another positive feature that arises from Uintah's component based architecture is the logical independence of the actual parallel implementation from the application. This gives the designer the ability to implement simulation on AMR grids with a high level of parallelism without the need to understand in depth the low-level parallel implementation, as depicted in Figure 2.9. Components, belonging to the area of expertise of the tuning expert, such as the simulation controller, the regridder, the load balancer, the data archiver and the scheduler are well isolated from those belonging to specific applications which are written and modified by experts in those specific domains.

Uintah implements domain-based parallelism: the computational grid is divided into hexahedral mesh patches and each of them is associated to a different process. In this scheme, the designer has to focus on the single patches: to her/him, in fact, it is required only to specify the solution scheme as a sequential flow of a tasks to be executed on each patch.

Each task, which is meant to be executed at every timestep, has to be characterized by which variables (and relative states) the computation is dependent upon and which of them are actually modified during the computation, together with their respective stencils. Analysing this information, the UCF computes a directed acyclic task graph to control parallel communication and execution. For each time step the scheduler controls the task execution order following this graph, assigning the execution of a task over a patch to the core this patch belongs to, and managing communication between neighbour patches and different AMR levels.

When no mesh refinement is adopted, the scheduler is instantiated once at the beginning of the simulation and than invoked cyclically at every timestep. Conversely, the scheduler has to be built every time regridding is executed since this procedure affects patches and requires the load balancing to reassign patches to different cores.

In its initial design, Uintah has been used to run simulations efficiently on up to a few thousand cores. Memory usage, data storage and computational loadings were preventing it to scale to larger than 4000 cores [61]. Data structure design was limiting the scalability: as increasing of the number cores (and thus of patches), also increasing memory usage was increasing, reaching the limit of available resources. Strong and weak scalability has, since then, been improved up to 96K, 262K, and 512K cores on the NSF Stampede, DOE Titan, and DOE Mira, respectively [6, 68].

The modular architecture of the UCF makes it an excellent research platform, making it possible to modify, exchange and test components without modifying the other components.

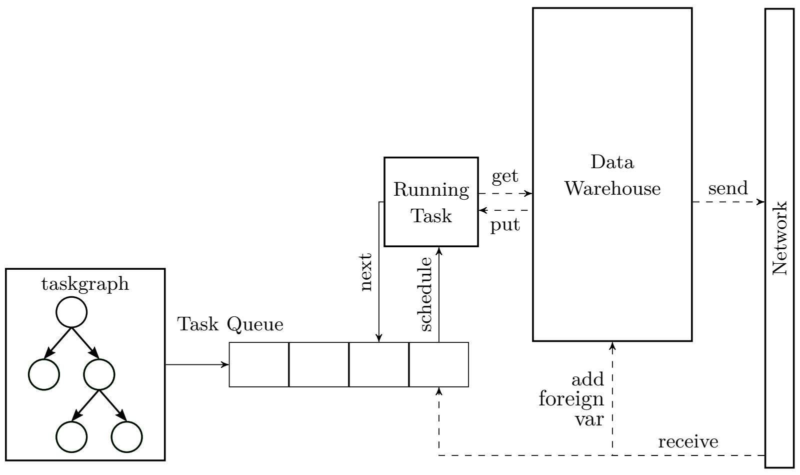


Figure 2.10: MPI Scheduler (edited from [62]).

This adaptability has already allowed use of Uintah across a wide range of applications: from the original problem of explosions in containers to haemodynamics of microvessels [35], analysis of foam subject to large deformation [10] and fire propagation [57].

Three main simulation components are available in Uintah at present: ICE, which implements an algorithm for both compressible and incompressible flows for systems of materials [52, 50, 51]; MPM, that is a particle in cell (PIC) method for solid mechanics on complex geometries [35] subject to large deformations [10] and fracture [36]; and MPMICE, which is the combination of the previous and is meant for fluid-structure simulations and overcomes the load imbalance that moving particles in MPM can cause when combined with ICE [61].

Many sub-components are also available within the UCF implementing various state equations, reaction models and constitutive laws.

2.2.2 Schedulers and Tasks

The *Task Scheduler* is the UCF component which controls the order in which tasks are executed and ensures that communications between processes are not corrupted. It analyses task and variables dependencies in order to be able to schedule the execution of each task in the correct sequence and builds two task graphs, one within the scope of the single process, and one for tasks operating between processes: the first is responsible for executing tasks in the *necessary order*, the latter for managing *required communications*.

Four main schedulers are available within Uintah:

Single Processor Scheduler which is designed to control the execution of tasks on a single processor. It doesn't use MPI and builds a static graph for the deterministic execution of tasks.

MPI Scheduler it is the basic scheduler and is designed to be used on distributed memory

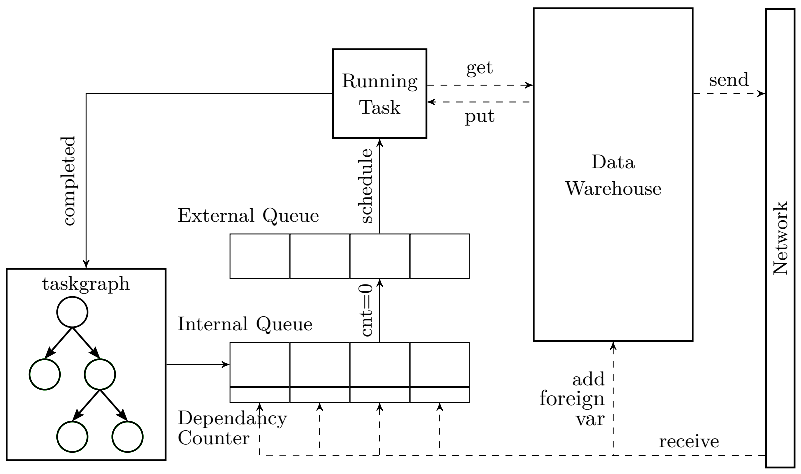


Figure 2.11: DynamicMPI Scheduler (edited from [62]).

architectures such as HPC clusters. A schematic representation of this scheduler is given in Figure 2.10. The model implemented by this schedule uses one process per core and to each process the same task-graph is distributed (*static task ordering*) which is executed in the same order (*deterministic execution*) on each core.

Dynamic MPI this scheduler overrides the limits of deterministic execution affecting the MPI scheduler. In such an execution scheme, in fact, a thread can sit idle while it is waiting for a message. This dynamic scheduler, as shown in Figure 2.11, can instead modify the order of execution of tasks to overlap communication and computation. This is achieved by generating two distinct task queues: one for internal and one for external tasks. Once a task has satisfied all of its internal dependency it is added to the internal queue where it waits for all MPI communications to complete. When the counter of its outstanding MPI messages reaches 0 it means that all communications have been completed, thus, the scheduler moves it to the external queue awaiting execution.

Unified Scheduler as depicted in Figure 2.12, this scheduler implements hybrid parallelism: it take advantage of the shared memory architecture of each computational node using pthreads for managing communications between processes within the same computational core and MPI for communications across different nodes. On each core there exists an access task queue which is shared among all the threads instantiated on that node. Within this paradigm, each thread, when it has completed its previous task, can assign to itself the next available task from the queue. This model can also take advantage of graphics processing units (GPUs), when available. In this case the scheduler builds a queue also for GPU tasks that can be accessed by the graphic units as the ordinary task queue is by normal CPU threads.

The approach implemented by the MPI Scheduler is well consolidated up to 100k cores but,

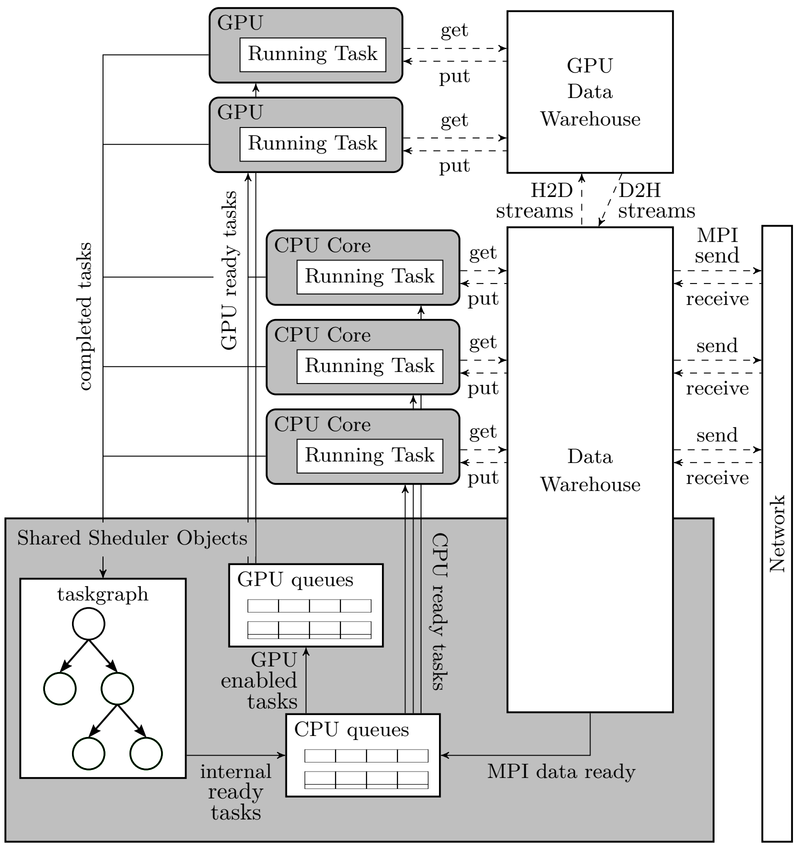


Figure 2.12: Multithreaded Unified Scheduler with GPU support (edited from [62]).

as the number of cores exceeds this limit, MPI communication time increases drastically due to memory usage associated with ghost cells and global metadata and becomes a barrier to scalability. On the contrary, the Unified Scheduler is demonstrated to overcome such a barrier assuring good scalability features up to 768k cores on the DOW Mira system [6].

2.2.3 Grid and Patches

Parallelism within Uintah is achieved by Patch-based domain decomposition. Each level of the computational grid is subdivided in patches which are assigned to available processors by the load balancer. Continuity of the solution, and when needed of its moments, among patches is imposed as boundary conditions on the single patches and the exchange of information between two adjacent patches is implemented via the introduction of ghost cells. When the MPI and the DynamicMPI schedulers are used, ghost cells are introduced around each patch since each patch is assigned to different processes each of which with its own individually assigned areas in memory. As a consequence, information between patches can be exchanged only via MPI

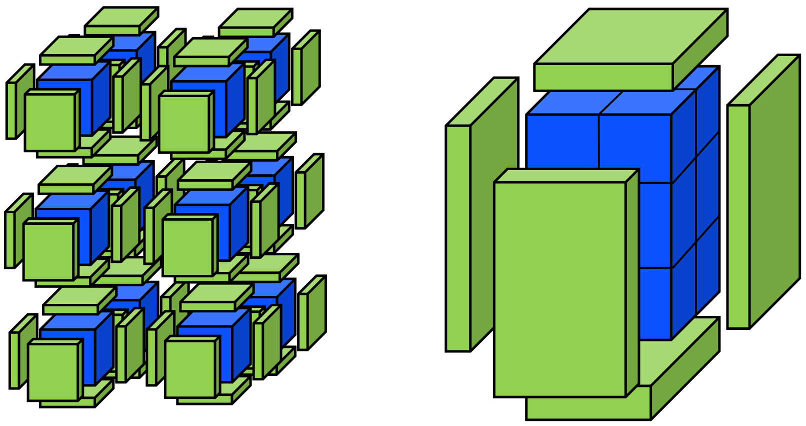


Figure 2.13: Ghost cells: memory usage is drastically reduced when hybrid-programming approach is adopted since the volume of ghost cells (green) is dropped when the volume of patch cells (blue) is kept constant.

messages. On the contrary, when the Unified scheduler is used, patches are grouped in clusters of adjacent patches which are assigned to the same node. In this architecture, each patch belonging to the same cluster is assigned to different threads on the same computational node. Since threads that belongs to the same same node can access the same shared region of memory, in this case, there is no need to create ghost cells between patches assigned to the same process, as shown in Figure 2.13. This hybrid programming approach drastically reduces, both, the amount of data that has to be stored in memory, and the size of MPI messages.

2.2.4 Data Warehouses and Variables

The variables in a physical model and its discretization can be of several types according to their spatial dependency and to which element of the grid their discrete values are associated. Uintah supports the following type of variables

`GridVariable` — this kind of variable is used to store the discretization of physical fields and any other function which is dependent on the spatial position. Depending on the grid element to which the discrete values are associated one of the following variable type derived from `GridVariable` can be used: `CCVariable`, for cell-centred discretizations; `NCVariable` for vertex based ones; or one of `SFCXVariable`, `SFCYVariable` and `SFCZVariable` for staggered face based discretizations.

`PerPatch` — this type of variable can be used to store information that are shared by all grid entries in a given patch, or that are associated to the patch itself.

`SoleVariable` — this kind of variable is used to handle values that are level specific, like functions of the grid refinement.

`ReductionVariable` — this kind of variable stores global quantities, like the time during the simulation or the norm of a field.

All of these variable types are templated on the data type they store; for example scalar fields are usually represented as `CCVariable<double>`, and the current timestep is of type `Reduction-Variable<int>`.

Variables are then stored in *Data Warehouses*, which collect together all the variables associated to the same stage of the simulation: there is a datawarehouse associated to the previous timestep and one to the current. Additional data warehouses whenever the grid is updated so that data associated to both the old grid and the new grid can be accessed.

When variables are accessed or modified by an application task a query is made to retrieve their data from the right datawarehouse for a given spatial range. If this range is within the patch on which a task is executed, accessing the datawarehouse is a straightforward operation since the data to retrieve belongs to the same processor executing the task. If the range extends beyond the path boundaries the query assumes that the information that may belong to other processors is already available within the datawarehouse because MPI communications are expected to be performed before the task is added to a scheduler queue. In fact, when an application schedule a task it must specify all task dependencies and computes, that is a list of the variables that are accessed, computed or updated by a given task together with the thickness of the ghost layer. This allows the scheduler to complete any MPI communication required by a task before executing it without exposing any of the communication complexity to the application itself.

AMR Grids

Within Uintah it is possible to compute multiple levels of mesh refinement, both adaptively and non-adaptively. In the latter case, meshes are created statically, from the coarsest to the finest, at the beginning of the simulation prior to solving any timestep. In the first case, conversely, a refinement flag is computed before each timestep and, depending on its values over the computational domain, the Regridder component decides whether to update or not the different levels of the grid.

The implementation of the task responsible for updating the refinement flag is left to the developer allowing maximum flexibility of the framework. The refinement flag is intended to be set to `true` on the cells where the approximation error is greater than a desired value.

The following input options controls the behaviour of the default Regridder component:

- $M \in \mathbb{N}$, the maximum number of levels;
- $\boldsymbol{r}^l \in \mathbb{N}^3$, the cell refinement ratio vector for each level $l = 0, \dots, M-1$;
- $\boldsymbol{s} \in \mathbb{N}^3$, the cell stability dilatation vector;
- $\boldsymbol{r} \in \mathbb{N}^3$, the cell regridding dilatation vector;

– $\boldsymbol{b} \in \mathbb{N}^3$, the minimum number of cell between one level's coarser level and its finer level.

At the beginning of the simulation, the coarsest level of the grid is computed following the specifications in the input file and a natural indexing of the cells is introduced, where indices are triplets of integers indicating the ordering of each cell along each direction. Then, a local error estimation is performed in order to flag which cells need to be regridded. Subsequently, the region of flagged cells is padded according to the cell stability and regridding dilatation vectors: the stability dilatation vector defines the number of cells that are used for padding in each direction around the flagged cell region with the purpose of enhancing the stability of the solution (such padding occurs at every timestep); the regridding dilatation vector controls the padding width with the purpose of reducing the frequency of regriddings (such padding occurs only before the regridding algorithm is invoked).

After all required levels have been generated, the regridder checks, for every level but the finest and the coarsest, that there exists a minimum number of cells between the boundaries of its coarser and finer levels as specified by $\boldsymbol{b}$, i.e., it checks that, along the x direction, there are at least b_x cells in level 2 that lie between the boundary of level 1 and that of level 3.

In addition, it is possible to control how often the regridding occurs by imposing the minimum and maximum number of timesteps that can occur between two regriddings (otherwise the refinement flag is updated after each timestep and regridding can occur before each timestep).

2.3 Numerical Techniques

In this section a brief overview of the numerical techniques adopted in this research work is given. First, a simpler reference model is introduced to simplify the following presentation. How to use finite-differences approximations to discretize a model both in space and in time is described. The resulting discrete model may be explicit or require the resolution of an implicit equation which may be solved using an implicit solver. In the last subsection the linear solvers adopted are briefly presented.

2.3.1 Model Problem: Heat Equation

The simplest time dependent partial differential equation is the heat diffusion model

$$\partial_t u(\boldsymbol{x}, t) = \alpha \Delta u(\boldsymbol{x}, t), \qquad \boldsymbol{x} \in \Omega,\, t > 0\,, \tag{2.25}$$

where $\alpha > 0$ is the constant diffusion coefficient.

This is completed with a generic initial condition $u(\boldsymbol{x}, 0) = v(\boldsymbol{x})$. In addition to periodic boundary conditions on rectangular domains, on more generic domains, it is common to impose a combination of Dirichlet boundary conditions $u_{|\Gamma_D} = g$ and Neumann conditions $\partial_{\boldsymbol{n}} u_{|\Gamma_N} = r$, where Γ_D and Γ_R form a disjoint partition of the domain boundary $\partial\Omega = \Gamma_D \uplus \Gamma_N$. Whenever g or r is constantly null, such conditions are said homogeneous.

2.3.2 Spatial Discretization

For the sake of simplicity, let $\Omega \subseteq \mathbb{R}$ be the a compact, $\Omega = [0, L]$, for which it is possible to chose a *grid step*, h, such that $L = M\,h$, with M a positive integer. In this way it is straightforward to introduce the grid with the following nodes (or vertices)

$$x_i = ih.$$

The field u is then identified with its values at the grid nodes

$$u_i(t) = u(x_i, t).$$

The next step is to approximate the differential operators in (2.25) with finite differences. The most of common finite difference formulas for the first order derivative are listed hereafter:

$$[D_x u]_{i+\frac{1}{2}} = \frac{u_{i+1} - u_i}{h}\,, \tag{2.26a}$$

$$[D_{2x} u]_i = \frac{u_{i+1} - u_{i-1}}{2h}\,, \tag{2.26b}$$

$$[D_x^- u]_i = \frac{u_i - u_{i-1}}{h}\,, \tag{2.26c}$$

$$[D_x^+ u]_i = \frac{u_{i+1} - u_i}{h}\,. \tag{2.26d}$$

Formulae (2.26a,2.26b) are both symmetric with respect to the point where the finite-difference is evaluated: Eqn. 2.26b involves only grid nodes, but it spans over three of them, while (2.26b) spans on two adjacent nodes but is associated to their midpoint $x_{i+\frac{1}{2}}$. Also (2.26c,2.26d) span on two adjacent nodes but the finite-difference is associated to one of the two. This avoids the introduction of the midpoint, but makes the formulae non-symmetric. The standard finite-difference approximation of the second order derivative is obtained applying (2.26a) twice

$$[D_t D_t u]^n = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}\,. \tag{2.26e}$$

If the field u is sufficiently regular, it is possible to write its Taylor series and compute the leading order error that is associated with each of the finite-difference above.

$$[D_x u]_{i+\frac{1}{2}} - u_x = \frac{1}{24} u_{xxx}(x_i) h^2 + \mathcal{O}h^4 \tag{2.27a}$$

$$[D_{2x} u]_i - u_x = \frac{1}{6} u_{xxx}(x_i) h^2 + \mathcal{O}h^4 \tag{2.27b}$$

$$[D_x^- u]_i - u_x = -\frac{1}{2} u_{xx}(x_i) h + \mathcal{O}h^2 \tag{2.27c}$$

$$[D_x^+ u]_i - u_x = \frac{1}{2} u_{xx}(x_i) h + \mathcal{O}h^2 \tag{2.27d}$$

$$[D_x D_x u]^n - u_{xx} = \frac{1}{12} u_{xxxx}(x_i) h^2 + \mathcal{O}h^4 \tag{2.27e}$$

When (2.26a) is used to approximate the Laplacian in 1D the following system of semi-discrete equation is obtained for problem over the grid nodes (2.25)

$$\partial u_t(x_i,t) = \alpha\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}, \qquad i=0,\dots,M,\ t>0 \tag{2.28}$$

which involves values of the solution – u_{-1}, u_{M+1} that are associated to *ghost nodes* outside of the computational domain.

It is possible also to associate the discrete values of u to the sub-intervals of size h identified by the grid nodes. These intervals are usually called cells, and in this case the discrete value of u are associated to cell midpoints. Cells are indexed by the index of their lower end. Therefore, in this case, $u_i(t) = u(x_{i+\frac{1}{2}},t)$ and the semi-discrete system obtained differs from the nodal case only in the range for i: $i=0,\dots,M-1$. In this case ghost values are introduced at $x_{-\frac{1}{2}}$ and $x_{M+\frac{1}{2}}$.

Boundary Conditions The value of the solution at ghost nodes can be extrapolated according to the boundary condition imposed on each end of the domain.

If the domain is periodic, the ghost nodes, x_{-1} and x_{M+1} can be identified with x_{M-1} and x_1 respectively. It is possible therefore to define

$$u_{-1} := u_{M-1} \qquad\qquad u_{M+1} := u_1$$

and the semi-discrete equations for $i=0$ and $i=M$ become identical. When the problem is cell centred, instead, the midpoint of the ghost cells, $x_{-\frac{1}{2}}$ and $x_{M+\frac{1}{2}}$ can be identified with x_M and x_0 respectively, which will not make identical ant semi-discrete equation.

When Dirichlet boundary condition are imposed to an end, it is possible to extrapolate the ghost value prolonging the derivative of u out of the computational domain. For example, in the cell based case at the lower end of the domain, imposing

$$\frac{u_0-u(x_0)}{h/2} = \frac{u_0-u_{-1}}{h/2},$$

and $u(x_0)=g$ gives the following expression for the ghost value

$$u_{-1} = g - 2u_0\,.$$

Ghost values for Neumann conditions are computed, instead, by imposing the value of the discrete derivative. For example, in the vertex based case at the higher end of the domain, imposing

$$\frac{u_{M+1}-u_M}{h} = r\,,$$

and $u(x_M)=r$ gives

$$u_{M+1} = u_M + hr\,.$$

Temporal Discretization

In order to fully discretize (2.28), or its counterpart for the cell centred case, a time step $k > 0$ is chosen so that the problem is solved at the discrete times $t^n = nk$. The notation

$$u_i^n \approx u_i(t^n)$$

is introduced for the discrete solution which may be different from the analytical solution because it is not possible any more to write $u_i^n = u(x_i, t^n)$ since we approximate the temporal derivative in (2.28) with finite difference.

Forward Euler When the forward finite difference (2.26d) is used, the following fully discrete system is obtained:

$$\frac{u_i^{n+1} - u_i^n}{k} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}, \qquad i \in \mathscr{I},\ n = 0, \dots, \tag{2.29}$$

where the set of spatial indices, $\mathscr{I}$, is chosen accordingly to the *variable base* (cell centred, or vertex based).

Supposing that the discrete solution, $\boldsymbol{u}^n = (u_i^n)$, at the n-th timestep is known, it is possible to write an explicit expression for computing the solution at the next timestep which depends only the values of $\boldsymbol{u}^n$:

$$u_i^{n+1} = u_i^n + k\alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}, \qquad i \in \mathscr{I},\ n = 0, \dots . \tag{2.30}$$

For this reason, method (2.30) is called Explicit Euler or Forward Euler. This method has the advantage to provide an explicit formula to compute the discrete solution $\boldsymbol{u}^{n+1}$ recursively from the discrete initial condition $\boldsymbol{u}^0 := (v(x_i))$.

The approximation of (2.25) with the Explicit Euler method, introduces a truncation error that is related to the approximation of the differential operators and a solution error, or simply *error*, that takes into account the differences between the numerical discrete solution and the true solution u that build up at every time iteration of the method. The first error can be estimated supposing that the known solution $\boldsymbol{u}^n$ is the exact solution $u_i^n = u(x_i, t^n)$ and its order in terms of the discretization parameters, h and k, can be computed using the leading order errors (2.27). This method truncation error is found to be linear in time, $\mathscr{O}(k)$, and quadratic in space $\mathscr{O}(h^2)$. The solution error for this method, however, is not guaranteed to vanish as the discretization steps are reduced, $h, k \to 0$; a von Neumann stability analysis [14] finds that the error is bounded over time (the norm of the discrete solution does not explode) only if the following condition is satisfied

$$r = \frac{\alpha k}{h^2} \leq \frac{1}{2}\,,$$

and for this reason this method is said conditionally stable.

Backward Euler When the forward finite difference (2.26c) is used to fully discretize (2.28), the following fully discrete system is obtained:

$$\frac{u_i^{n+1} - u_i^n}{k} = \alpha \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{h^2}, \qquad i \in \mathscr{I},\ n = 0, \dots, \tag{2.31}$$

where the set of spatial indices, $\mathscr{I}$, is chosen accordingly to the variable base.

In this case it is not possible to write an explicit expression for computing the solution at the next timestep which depends only the values of $\boldsymbol{u}^n$. In this case the fully discrete system can be expressed as the following linear system

$$A\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \boldsymbol{b}, \qquad n = 0, \dots,, \tag{2.32}$$

where A is a tri-diagonal matrix with diagonal entries equal to $1 + \frac{2\alpha k}{h^2}$ and off-diagonal entries equal to $-\frac{\alpha k}{h^2}$, and $\boldsymbol{b}$ takes into account for the boundary conditions. This system must be solved at every time iteration and, for this reason, this is an *implicit method* and is called Implicit Euler or Backward Euler. This method truncation error is linear in time, $\mathscr{O}(k)$, and quadratic in space $\mathscr{O}(h^2)$, like the Explicit Euler method, but has the advantage to be *unconditionally stable.*

Crank Nicholson When the centred finite difference (2.26a) is used to fully discretize (2.28), and the mid-point solution $u^{n+\frac{1}{2}}$ is defined as the average between the previous and the next timestep solution

$$u_i^{n+\frac{1}{2}} := \frac{u_i^n + u_i^{n+1}}{2}$$

the fully discrete system can be expressed as a linear system

$$A\boldsymbol{u}^{n+1} = \boldsymbol{v}^n + \boldsymbol{b}, \qquad n = 0, \dots,, \tag{2.33}$$

where the matrix A is tri-diagonal with diagonal entries equal to $1 + \frac{\alpha k}{h^2}$ and off-diagonal entries equal to $-\frac{\alpha k}{2h^2}$, and $\boldsymbol{v}^n = \alpha k \Delta u^n$ (with Δ being the discrete Laplacian operator, the tri-diagonal matrix with diagonal values equal to $-\frac{2}{h^2}$ and off-diagonal values equal to $\frac{1}{h^2}$) and $\boldsymbol{b}$ takes into account for the boundary conditions. This implicit method is called Crank-Nicholson ant its truncation error is quadratic both in time and space, $\mathscr{O}(k^2 + h^2)$ and is unconditionally stable like the Backward Euler method.

2.3.3 Linear Solvers

The implicit time stepping methods introduced in the previous section – Backward Euler and Crank Nicholson – require that a linear system is solved to compute the current time step solution from the previous one. hereafter we briefly present the linear solver used in this work. This is not an exhaustive list of possible linear solvers, but covers the linear solvers provided by the HYPRE library which is used by Uintah.

For the sake of simplicity, in this section we consider the following generic linear system:

$$A\boldsymbol{x} = \boldsymbol{b}, \qquad A \in \mathbb{R}^{m\times m},\ \boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^m.$$

To reduce the condition number of A it is common practice to resort to a preconditioner P, that is a non singular matrix that is simple enough to invert and solve the following equivalent linear system

$$P^{-1}A\boldsymbol{x} = P^{-1}\boldsymbol{b}$$

Ideally the condition number of $P^{-1}A$ should be uniformly bounded in m, so that the number of iterations required by the following iterative methods to converge is bounded as the matrix order increases. This will allow the solver to scale well as the problem dimension/number of processes is increased.

A simple preconditioner is provided by the diagonal matrix

$$P = \begin{bmatrix} a_{00} & 0 & \cdots & 0 \\ 0 & a_{11} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{mm} \end{bmatrix}$$

which is called the *Jacobi preconditioner*.

Even though the choice of the best preconditioner is crucial in controlling the performance of linear solvers, little focus has been given to them in this research work. The reason behind this, is that the main focus of this research is the implementation of a scalable software for solving phase-field problems, therefore only support for preconditioners has been developed. The analysis of which particular preconditioner is the most suitable for any of the applications considered has been left to future work since it is beyond the remit of this work.

The scalability performance of the solvers listed below is assessed in the following chapters taking into account the resources required by each method with either no preconditioner or Jacobi's. The performance of the solver using few iterations of another method as preconditioner will then be a combination of the performance of the two methods as unpreconditioned solver.

GMRES The General Minimal Residual Method [83] is an iterative *Krylov solver*. The n-th Krylov space is the subspace generated by the images of the initial error[2], $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$, under the first n powers of A:

$$\mathcal{K}_n = \left\langle A^k \boldsymbol{r}_0 \right\rangle_{k=0}^{n-1} .$$

The first step of the algorithm is to compute an orthonormal basis, $\boldsymbol{q}_k \in \mathbb{R}^m$ $(k = 0, n-1)$, for K_n and a matrix $H_n \in \mathbb{R}^{n \times n+1}$ such that

$$AQ_n = Q_{n+1} H_n ,$$

where $Q_n \in \mathbb{R}^{m \times n}$ is the juxtaposition of the first n $\boldsymbol{q}_k$ base vectors. These are computed using the Arnoldi's method [2]. Introducing the vector $\boldsymbol{y}_n := Q_n^{-1}(\boldsymbol{x}_n - \boldsymbol{x}_0)$ it can be shown that the n-th residual, $\boldsymbol{r}_n := \boldsymbol{b} - A\boldsymbol{x}_n$ can be written as $\boldsymbol{r}_n = H\boldsymbol{y}_n - \|\boldsymbol{r}_0\| \mathrm{e}_0$. The second step of the GMRES iteration is therefore to compute the $\boldsymbol{y}_n$ which minimizes $\boldsymbol{r}_n$ solving a linear least square problem of size n. If the residual is small enough then the solution is computed as $\boldsymbol{x}_n = \boldsymbol{x}_0 + Q_n \boldsymbol{y}_n$. Otherwise n is increased by one and the algorithm is reiterated.

LGMRES When the dimension of the Krylov space increases, the memory requirement of GMRES grows since the number of required vectors increases. Also the number of multiplication is $\frac{1}{2}mn^2$. To avoid this difficulties, the GMRES method can be restarted every k iterations, being k a fixed integer parameter. The restarted GMRES present some problems in the converges, due to alternating residual vectors. The LGMRES method [4] is designed to avoid the slowing of convergence in restarted GMRES.

FlexGMRES The Flexible Inner-Outer Preconditioned GMRES Algorithm [82] is a variant of the GMRES algorithm. It allows to change preconditioner at every step. Thanks to this flexibility, any iterative method could be used as a preconditioner. An even more interesting application is using FlexGMRES together with relaxation techniques such as those of multilevel solver.

BiCGStab The Stabilized Bi-Conjugate Gradient method [92], is a modification of the Bi-Conjugate Gradient (BiCG) method [31] which converges more smoothly than the latter. As for the BiCG method, the idea to generalize the Conjugate Gradient (CG) method [39] is to build two different sets of residual vector, $\boldsymbol{r}^k$ and $\tilde{\boldsymbol{r}}^k$, such that $\boldsymbol{r}^{\bar{k}}$ is orthogonal to any $\tilde{\boldsymbol{r}}^k$ with $k = 0, \ldots, \bar{k} - 1$ and vice versa.

Hybrid This method provides a tool to switch between the diagonally scaled GMRES solver (using the simple Jacobi preconditioner) and a GMRES solver using a more complex and more

[2]when no guess is given $\boldsymbol{x}_0$ is chosen to be $\boldsymbol{0}$ and $\boldsymbol{r}_0 \equiv \boldsymbol{b}$

expensive preconditioner. It starts with diagonally scaled solver and monitors how fast the Krylov solver converges to switch to the other when there is not sufficient progress.

CycRED The Cyclic Reduction method [32], contrarily to the previous ones, in not an iterative method. The method proceeds by splitting the problem separating different rows and columns in multiple sub-problems. If the linear matrix A is a block Toeplix matrix, as it is the case for the heat problem, the collection of resulting sub-problems are correspond to 1D tri-diagonal systems which can be solved cheaply by a direct method.

SMG The parallel Semi-coarsening Multigrid Solver [88, 9, 28] is a multigrid solver for the linear systems arising from the discretization of the diffusion equation on rectangular grids. The basic idea of multigrid solvers it to speed up the convergence of iterative methods avoiding the low frequencies component of the error by changing the grid. Low frequency smooth errors which hold back convergence can be solved on coarser grids, while high frequency errors that are not visible on a coarse grid are solver on finer grids. The multigrid levels, generally, do not correspond to the AMR levels: in the first case, levels are introduced to improve the speed of an iterative method, in the latter, they are introduced to reduce the size of the problem to solve.

Let us consider a family of discrete linear problems

$$A_j \boldsymbol{x}_j = \boldsymbol{b}_j, \qquad j = 0, \dots, M$$

where the index M corresponds to the linear system we want to solve. The j-th linear problem can be regarded as the discretization of a given problem on a spatial grid with spacing $h_j = 2h_{j+1}$ with $h_M = h$ the finest (highest) level. For each level we consider a generic preconditioned iterative method

$$\boldsymbol{x}_j^{k+1} = \boldsymbol{x}_j^k + P_j(\boldsymbol{b}_j - A_j \boldsymbol{x}_j^k) \,,$$

where P_j is the preconditioner for the j-th problem. The error at the k-th iteration on the j-th level can be written as

$$\boldsymbol{e}_j^k = B_j \boldsymbol{e}_j^k \,,$$

where $B_j := \mathbb{I} - P_j A_j$.

Generally, the multigrid algorithm performs n_{pre} times a *pre-smoothing* step of the initial guess $\boldsymbol{x}_j^0$ on the finest grid ($j = M$)

$$\boldsymbol{x}_j^k = \boldsymbol{x}_j^{k-1} + P_j(\boldsymbol{b}_j - A_j \boldsymbol{x}_j^{k-1}), \qquad k = 1, \dots, n_{\text{pre}} \,.$$

Then, the *coarse grid correction* step is performed, which involves three sub-steps:

1. *restrict* the residual $\boldsymbol{r}_j := \boldsymbol{b}_j - A_j x_j^{n_{\text{pre}}}$ on the coarse grid

$$\boldsymbol{r}_{j-1} = R_j^{j-1}\boldsymbol{r}_j$$

   being R_j^{j-1} an opportune restriction operator which maps a discrete fields on the j-th (fine) grid to a field on the $(j-1)$-th (coarse) grid;

2. solve the *defect problem*:

$$A_{j-1}\boldsymbol{x}_{j-1} = \boldsymbol{r}_{j-1}$$

3. correct the solution

$$\bar{\boldsymbol{x}}_j^0 = \boldsymbol{x}_j^{n_{\text{pre}}} + \Pi_{j-1}^j \boldsymbol{x}_{j-1}$$

   where Π_{j-1}^j is an opportune prolongation operator which maps a discrete fields on the $(j-1)$-th (coarse) to a field on the grid j-th (fine) grid.

Finally, the multigrid algorithm performs n_{post} times a *post-smoothing* step on $\bar{\boldsymbol{x}}_j^0$ on the finest grid $(j = M)$

$$\bar{\boldsymbol{x}}_j^k = \bar{\boldsymbol{x}}_j^{k-1} + P_j(\boldsymbol{b}_j - A_j\bar{\boldsymbol{x}}_j^{k-1}), \qquad k = 1, \ldots, n_{\text{post}} .$$

This basic multigrid algorithm is referred to as *V-cycle*. Multiple V-cycles can be applied recursively, which increases the number of level of the method.

SMG is a particularly robust method. The algorithm semi-coarsens in the highest order direction to obtain a problem defined on a lower dimension grid. This problem is solver using the SMG method until the resulting problem is defined over a 1D grid. This inner problem is then resolved with one V-cycle.

PFMG, SysPFMG The PFMG solve [3, 28] is a parallel semi-coarsening multigrid solver similar to SMG which is much more efficient per V-cycle but less robust than SMG.
The SysPFMG solver, is a generalization of the PFMG solver for system of elliptic PDEs.

FAC The FAC solver is a parallel Fast Adaptive Composite grid solver, which implements a multigrid solver on complex grid such as AMR grids. For details of the basic overall algorithms, see [64].

Split The Split solver is another solver for composite girds. It is a parallel block Gauss-Seidel method [33] which, for problems with only one variable, can be viewed as a domain-decomposition solver without overlapping. Each grid part is solved performing either a SMG or a PFMG V-cycle.

Chapter 3

Development Approach and Challenges

3.1 Development Approach

The implementation of a Phase Field solver within the Uintah Computational Framework has followed an incremental approach. Features have been added to the implementation one at the time.

3.1.1 PureMetal Solver

The first software being developed during this research is a node-centered sequential implementation of an explicit solver for the Pure Metal problem (2.5).

This implementation, which is presented in §4.1, is a standalone software, called PureMetal, which doesn't use Uintah. Its purpose is to have an initial implementation and in its development the fully discretized equations for the model are introduced. For simplifying these equations, auxiliary anisotropic quantities are defined whose computation will represent one of the tasks in the following implementation within Uintah.

The post-processing of the output of this initial PureMetal implementation is then performed using Matlab.

3.1.2 PhaseField Initial Component

The second step in the development is the implementation of the PhaseField Uintah Component described in §4.2. This step is articulated in three phases. In the first of these phases, parallelism is achieved taking advantage of the Uintah Computational Framework. This is done by developing two distinct components for cell-centred and node-centred explicit solvers of the pure metal model in two dimensions. Fixed Dirichlet or Neumann/Dirichlet boundary conditions are imposed depending on the computational domain: the first ones are used when the problem is solved over the full computational domain, while the second conditions are used if symmetry has been taken advantage of to quarter the computational domain.

The second phase is to use the data structures and methods provided by Uintah to implement adaptive mesh refinement. For these initial Uintah components a naive approach is chosen to maximally simplify this phase: only two grid levels are introduced with the assumption that the fine level is wide enough that the solution evolves only within this level, so that no fine/coarse interface condition has to be implemented at this stage.

In the third phase, effort has been given to extend the cell-centered and node-centered component to solve pure metal model in three dimensions and to implement fine/coarse interface conditions. However these efforts stressed out the limitations in the implementation of the current approach which proved to be too prone to errors.

3.1.3 Heat Component

To have a simpler starting point on top of which implement and test new features, also a Heat component has been developed §4.3. That is because the heat problem (2.25) is the simplest elliptical problem with an explicit analytical solution.

Two different development paths branched from this component: an early implementation of an implicit solver which served the purpose of understanding how Uintah implements its support to HYPRE and its linear solvers and preconditioners; and the development of more general implementation of the finite difference approximations that could be used to quicker implement solvers for multiple applications that are also more efficient.

The progress made in these two directions resulted in a redesign of the Heat Component which provided additional features and resulted to be a general finite difference development framework within Uintah: a new PhaseField component with multiple applications for solving different problems.

3.1.4 View Framework

The first step in the direction of providing a more general finite-difference framework has been the implementation of the backbone of the framework itself as described in Chapter 5. In this chapter, the concepts of view (`View`) and finite-difference view (`FDView`) are introduced. They are interfaces to be used by application developers to access finite difference implementations through high level classes and methods.

A great advantage of the view concepts is that the same kernel can be implemented for each task of an application to update the solution on a patch regardless of its topological characteristics and of those of its neighbour patches. The view interfaces successfully take away from the application developer the responsibility to distinguish internal and boundary patches.

It also hide effectively from him/her any complexity in handling different boundary or interface conditions that may be applied.

In the first step of the implementation, as presented in Chapter 5, boundary conditions and fine coarse interfaces are not yet implemented, but the great flexibility introduced by the new approach is already evident in the applications for solving the proposed benchmark problems on periodic domains. The results in §5.3 show how the explicit solvers implemented as applications of the new PhaseField component can solve second-order Allen-Cahn and fourth-order Cahn-Hilliard problems as long as these problem are not too stiff to be explicitly solved.

3.1.5 Boundary Conditions

The second step in the development of the general framework is the implementation of generic boundary conditions.

This is achieved thanks to three features. First, the definition of as many implementation of the view interfaces as the possible combinations of boundary conditions and faces on which such conditions must be applied. These boundary views are introduced in Chapter 6 using class templates and meta-programming techniques so that these combinations are automatically created by the compiler using implementation of the discretised boundary conditions that are independent of the face and direction on which these are applied. This aspect is fundamental in limiting code duplication and errors while implementing new boundary conditions.

The second feature is the introduction of a partitioner algorithm to detect which portions of each patch can share the same view implementation. Thanks to the partitioner the different view implementations are instantiated at the beginning of a simulation or when the geometry is updated so that at each timestep the solution can be computed iterating over the partitioned regions instead of over the patches. This change is key in avoiding wasting computational time in looking up for the implementation appropriate to each region.

The third feature is the introduction of the Problem structure as the container that holds at the same time all the information about each region identified by the partitioner and all the view implementation instances for that region.

Thanks to these three new features the application developer can loop over the list of Problems and use its view interfaces to evaluate the same kernel definition. The compiler will then create several instances of the kernel for each view implementation which is specific to that view implementation.

3.1.6 Adaptive Mesh Refinement

The third step in the development of the general finite difference framework provided by the PhaseField component is the implementation of Fine/Coarse Interfaces as described in Chapter 7. This step is made simple by the Uintah Computational Framework as well as by the previous step.

On one hand Uintah already provides for all the infrastructure for handling different grid levels and for the necessary regrid procedures, on the other hand the implementation of boundary conditions is general enough that fine/coarse interfaces can be implemented simply defining the generic view implementation for those regions identified by the partitioner to be the edge of a patch adjacent to a patch on a different grid level.

In addition to the new view implementations of the interface condition, special views have been defined to provide operators of interpolation and restriction between different levels. These special views are extensions of the view interfaces and take advantage of the view and problem framework so that also their definition is general and independent of the patch location and boundaries.

However, the choice of which fine/coarse interface condition is best suited for solving phase-field problems is less trivial. Great effort at this stage has been given in developing tools to evaluate the error introduced by fine/coarse interfaces and in identifying which regridding parameters provide the best compromise between accuracy and speed.

3.1.7 Implicit Solvers

While the general phase-field framework was being developed, work has been done also in the direction of implementing implicit and semi-implicit solvers. In particular, the first step in this direction has been to extend the Heat component to use the existing driver to HYPRE provided by Uintah. This driver provides a bridge between Uintah's structures and HYPRE's linear solvers and preconditioners on structured grids and as such does not support grids composed of multiple levels.

The first implementation of an implicit solver for the heat problem could solve only one level at a time. The coarser level is solved first, using the problem boundary conditions and the restriction of the solution at the previous time step as initial condition. Each finer level is then solved in sequence from the coarser to the finest using the solution at the coarser level as boundary conditions.

A second step in the development has been the implementation of a new driver to HYPRE's semi structured linear solvers. These solvers can handle geometries represented as the union of patches with different refinement steps and therefore can be used to update solutions over AMR grids.

Few changes where required to the view framework to support implicit solvers, mainly related to the fact that differential operators in this case must return coefficients for stencil entries of a linear matrix and a right-hand-side vector.

3.2 Challenges

Working with a large software package such as Uintah presents many challenges. First of all is the size of the code base itself. Over thirty years of development, the Uintah repository grew to count about 5,000 files with more than 500,000 non-blank lines of C++ code. C++ represents about 80% of the codebase, but other languages are present, such as Matlab, Python and Fortran.

Being Uintah a research software in continuous active development, most of the lower level code is either undocumented or has little or not updated documentation. However, the component based design and an homogeneous code style, simplify the task of understanding the codebase whose learning curve remains in any case very steep. It is reasonable to estimate that understanding how to work within Uintah could take about a year.

The second main challenge in working with such a big research software is that it is being actively developed and that other researcher can make changes to other components that affect the code you are currently working on. This problem was exaggerated for this research by two factors:

– the new component being developed required features and behaviours not shared by other applications of Uintah for which no continuous integration test was in place; and

– the revisioning approach adopted by Uintah being centralised, which resulted in a longer process of validation for the new code before it was included into the central repository.

During the whole research it has been very complicated to understand when to commit changes to the central repository and when to keep up with changes in the central repository. Recently, Uintah moved to a distributed version control system making much easier the process of developing new features.

The third main challenge has been the integration of HYPRE semi-structured solvers. This is another actively developed research software which presents similar challenges to Uintah. Despite implicit solvers and semi-structured grid were supported both by Uintah, they couldn't work together. In the process of integrating these two features, some changes in the workflow of Uintah have been necessary because new edge have cases being introduced. This integration required to understand the behaviour of most of the components of Uintah. Small changes had been necessary to the datawarehouse, the scheduler and also to the MPI messages handler to make the two feature integrate.

Chapter 4

Initial Implementations

In this first chapter about the implementations developed during this research work, focus is given to the early stages of development. This will help highlighting the most critical aspects in the development of a phase field component within the UCF.

Throughout the whole research, an incremental approach to the development of the component has been used: only one degree of complexity is added at a time. Each development stage is then tested before adding more complexity to the implementation. At first, a sequential code for the explicit solution of the 2D pure metal problem (2.5) has been developed, as described in §4.1. This initial sequential code has then been ported to Uintah as one of its components.

At this stage two path of development could have been followed: implementing an implicit time-stepping method, or implementing mesh adaptivity. To follow the first path, a simpler Uintah component has been developed in §4.3, the Heat component. This component, as discussed in §4.3, implements a solver for the heat diffusion problem (2.25). Being this problem the simplest elliptic partial differential equation, it has been extensively studied both analytically and numerically. It is therefore the most natural choice while developing an implicit time stepping algorithm, at least in the initial stage when the implementation of the liner solver has to be validated.

Following the second path (§4.2.3 and extending the component to three dimension problems (§4.2.5, instead, stressed out critical aspects of the implementation. These are discussed in §4.3.2 and will need an ad-hoc solution in Uintah, which is presented in the following chapters.

4.1 Pure Metal C++ Sequential Code

The first step in this research was to implement a simple software to simulate a solidification process governed by model (2.1) for an isotropic pure metal in two dimensions. For the sake of simplicity, a finite differences explicit Euler scheme has been chosen for the discretization of the problem. Subsequently, the anisotropy function has been added to the model while keeping the same discretization scheme.

4.1.1 2D Isotropic Model

Since, in this case, either W and τ are constant, given the definition (2.2) and setting[1] $h(\psi) := \psi$, the governing equations reduce to the following:

$$\begin{aligned} \tau\partial_t\psi &= W^2\nabla^2\psi + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_t u &= \alpha\nabla^2 u + \tfrac{1}{2}\partial_t\psi \ , \end{aligned} \qquad \text{on } \Omega \subset \mathbb{R}^N,\ \forall t > 0 \ . \tag{4.1}$$

These are completed by the set of initial conditions in (2.10), and boundary conditions can be chosen to be either Dirichlet,

$$\psi \equiv -1 \ , \qquad u \equiv -\Delta \ , \qquad \text{on } \Gamma_D \ ,$$

or homogeneous Neumann,

$$\partial_{\boldsymbol{n}}\psi \equiv 0 \ , \qquad \partial_{\boldsymbol{n}} u \equiv 0 \ , \qquad \text{on } \Gamma_N \ ,$$

where Γ_D and Γ_N is a disjoint partition of the boundary of the computational domain, $\Gamma_D \uplus \Gamma_N = \partial\Omega$.

Nondimensionalisation

Eqns. 4.1 can be nondimensionalised setting $\tilde{\boldsymbol{x}} := \frac{\boldsymbol{x}}{W}$, and $\tilde{t} := \frac{t}{\tau}$:

$$\begin{aligned} \partial_{\tilde{t}}\psi &= \tilde{\nabla}^2\psi + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_{\tilde{t}} u &= \tilde{\alpha}\tilde{\nabla}^2 u + \tfrac{1}{2}\partial_{\tilde{t}}\psi \ , \end{aligned} \qquad \text{on } \tilde{\Omega} \subset \mathbb{R}^N,\ \forall \tilde{t} > 0 \ ,$$

where the nondimensional thermal diffusivity is defined by $\tilde{\alpha} := \frac{\alpha\tau}{W^2}$. Since parameters α and λ are bonded together by (2.9), if kinetics is neglected by setting $\beta = 0$, it is possible to write $\lambda = \frac{\alpha\tau}{a_2 W^2} = \frac{\tilde{\alpha}}{a_2}$ (where $a_1 = 0.8839$ and $a_2 = 0.6267$ [49]). This leaves $\tilde{\alpha}$ to be the only parameter left to be chosen.

Time Discretization

Fix a timestep $\Delta t > 0$ and let $t^n := (\Delta t)^n$, then the forward Euler (explicit) time discretization of the problem is:

$$\Delta\psi^n = \Delta t[\Delta\psi^n + \psi^n(1-[\psi^n]^2) - \tfrac{\alpha}{a_2}u^n(1-[\psi^n]^2)^2]$$

[1]This choice corresponds to an isothermal variational formulation of the problem that cannot be derived from a single-Lyapunov functional but is more computational appealing [49].

$$\begin{aligned} \psi^{n+1} &= \psi^n + \Delta\psi^n \\ \partial_t u^{n+1} &= u^n + \alpha \Delta t\, \Delta u^n + \tfrac{1}{2}\Delta\psi^n\,, && n > 0\,, \end{aligned}$$

where the tilde has been omitted for the sake of simplicity. Defining $\psi^0 := \psi_0$ and $u^0 := u_0$, ψ^n and u^n are then, for all $n \geq 0$, approximations of the nondimensional temperature and phase fields at time t^n.

Space Discretization

Let $h > 0$ be the spacing of an uniform grid $\{(x_i, y_j) : i = 0, \dots, N_x, j = 0, \dots, N_y\}$ where $x_i := x_0 + ih$ and $y_j := x_0 + jh$. Naming $\psi^n_{i,j}$ and $u^n_{i,j}$ the approximations of the nondimensional temperature and phase fields at time t^n on the node (x_i, y_j) derivatives can be approximated by central differences:

$$\begin{aligned} \partial_x\psi^n(x_i,y_j) &\approx \tfrac{\psi^n_{i+1,j}-\psi^n_{i-1,j}}{2h}\,, & \partial^2_x\psi^n(x_i,y_j) &\approx \tfrac{\psi^n_{i+1,j}+\psi^n_{i-1,j}-2\psi^n_{i,j}}{h^2}\,, \\ \partial_y\psi^n(x_i,y_j) &\approx \tfrac{\psi^n_{i,j+1}-\psi^n_{i,j-1}}{2h}\,, & \partial^2_y\psi^n(x_i,y_j) &\approx \tfrac{\psi^n_{i,j+1}+\psi^n_{i,j-1}-2\psi^n_{i,j}}{h^2}\,. \end{aligned}$$

The numerical scheme for inner nodes then becomes:

$$\begin{aligned} \Delta\psi^n_{i,j} &= \Delta t[\tfrac{\psi^n_{i+1,j}+\psi^n_{i-1,j}+\psi^n_{i,j+1}+\psi^n_{i,j-1}-4\psi^n_{i,j}}{h^2} - \psi^n_{i,j}(1-[\psi^n_{i,j}]^2) - \tfrac{\alpha}{a_2}u^n_{i,j}(1-[\psi^n_{i,j}]^2)^2] \\ \psi^{n+1}_{i,j} &= \psi^n_{i,j} + \Delta\psi^n_{i,j} \\ u^{n+1}_{i,j} &= u^n + \alpha\Delta t\,\tfrac{u^n_{i+1,j}+u^n_{i-1,j}+u^n_{i,j+1}+u^n_{i,j-1}-4u^n_{i,j}}{h^2} + \tfrac{1}{2}\Delta\psi^n_{i,j}\,, \\ & \qquad 0 < i < N_x,\ 0 < j < N_y,\ \forall n > 0\,. \end{aligned}$$

How to compute field values at boundary nodes is straightforward in the case of Dirichlet conditions while homogeneous Neumann condition are implemented by the introduction of ghost nodes at $i = -1, N_x + 1$ and $j = -1, N_y + 1$ and defining:

$$\begin{aligned} \psi^n_{-1,j} &:= \psi^n_{1,j}\,, & \psi^n_{N_x+1,j} &:= \psi^n_{N_x-1,j}\,, & u_{-1,j} &:= u^n_{1,j}\,, & u^n_{N_x+1,j} &:= u^n_{N_x-1,j}\,, \\ \psi^n_{i,-1} &:= \psi^n_{i,1}\,, & \psi^n_{i,N_y+1} &:= \psi^n_{i,N_y-1} & u_{i,-1} &:= u^n_{i,1}\,, & u^n_{i,N_y+1} &:= u^n_{i,N_y-1}\,. \end{aligned}$$

4.1.2 2D Anisotropic Model

With the same choice of h and f and choosing formulation (2.7) for the anisotropy function, model (2.3) becomes:

$$\begin{aligned} \tau_0 A\partial_t\psi &= W_0^2\nabla\cdot(A^2\nabla\psi) + W_0^2\nabla\cdot\left[|\nabla\psi|^2 A\,\partial_{\nabla\psi}A\right] + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_t u &= \alpha\Delta u + \tfrac{1}{2}\partial_t\psi\,, && \text{on } \Omega \subset \mathbb{R}^N,\ \forall t > 0\,. \end{aligned}$$

Expanding the divergence operators in two dimensions, the following is obtain:

$$\begin{aligned}\tau_0 A\partial_t\psi &= W_0^2 A^2\Delta\psi + \nabla[A^2]\cdot\nabla\psi + W_0^2\partial_x\left[|\nabla\psi|^2 A\,\partial_{\partial_x\psi}A\right] + W_0^2\partial_y\left[|\nabla\psi|^2 A\,\partial_{\partial_y\psi}A\right] + \\ &\quad + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_t u &= \alpha\Delta u + \tfrac{1}{2}\partial_t\psi \qquad\qquad \text{on } \Omega\subset\mathbb{R}^N,\ \forall t>0\ .\end{aligned}$$

Observing that

$$\partial_{\psi_\alpha}A = \begin{cases}\partial_{\boldsymbol{n}}A\cdot\partial_{\psi_\alpha}\boldsymbol{n} = \partial_{\boldsymbol{n}}A\cdot\left(\frac{\boldsymbol{e}_\alpha}{|\nabla\psi|} - \partial_\alpha\psi\frac{\nabla\psi}{|\nabla\psi|^3}\right)\ , & \text{for } |\nabla\psi|\neq 0 \\ 0\ , & \text{otherwise}\end{cases} \qquad \alpha = x, y\ ,$$

it is possible to write:

$$\begin{aligned}\tau_0 A\partial_t\psi &= W_0^2 A^2\Delta\psi + \nabla[A^2]\cdot\nabla\psi + W_0^2\partial_x\Big[\tfrac{A\partial_{\boldsymbol{n}}A}{|\nabla\psi|}\cdot\Big([\partial_y\psi]^2,\ -\partial_x\psi\,\partial_y\psi\Big)^t\Big] + \\ &\quad + W_0^2\partial_y\Big[\tfrac{A\partial_{\boldsymbol{n}}A}{|\nabla\psi|}\cdot\Big(-\partial_x\psi\,\partial_y\psi,\ [\partial_x\psi]^2\Big)^t\Big] + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_t u &= \alpha\Delta u + \tfrac{1}{2}\partial_t\psi\ , \qquad\qquad \text{for } |\nabla\psi|\neq 0\ ,\end{aligned}$$

which leads to the following system:

$$\begin{aligned}\tau_0 A\partial_t\psi &= W_0^2 A^2\Delta\psi + \nabla[A^2]\cdot\nabla\psi + W_0^2\partial_x\left[\partial_y\psi\tfrac{A}{|\nabla\psi|}(\partial_{\boldsymbol{n}_x}A\partial_y\psi - \partial_{\boldsymbol{n}_y}A\partial_x\psi)\right] + \\ &\quad + W_0^2\partial_y\left[\partial_x\psi\tfrac{A}{|\nabla\psi|}(\partial_{\boldsymbol{n}_y}A\partial_x\psi - \partial_{\boldsymbol{n}_x}A\partial_y\psi)\right] + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_t u &= \alpha\Delta u + \tfrac{1}{2}\partial_t\psi\ , \qquad\qquad \text{for } |\nabla\psi|\neq 0\ ,\end{aligned}$$

The following functions of the normal vector are introduced

$$B_{\alpha,\beta} := \tfrac{A}{|\nabla\psi|}(\partial_{\boldsymbol{n}_\alpha}A\partial_\beta\psi - \partial_{\boldsymbol{n}_\beta}A\partial_\alpha\psi)\ , \qquad \alpha,\beta\in\{x,y\}\ ,$$

which are extended to value 0 where $|\nabla\psi| = 0$.

Using this definition, observing that $B_{\alpha,\beta} = -B_{\beta,\alpha}$, and apply the Leibniz rule to the derivatives of the products $\partial_\alpha\psi B_{\beta,\alpha}$, the previous system simplifies to:

$$\begin{aligned}\tau_0 A\partial_t\psi &= W_0^2 A^2\Delta\psi + \nabla[A^2]\cdot\nabla\psi - W_0^2\,\partial_x\psi\,\partial_y B_{x,y} + \\ &\quad + W_0^2\,\partial_y\psi\,\partial_x B_{x,y} + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\ \partial_t u &= \alpha\nabla^2 u + \tfrac{1}{2}\partial_t\psi\ , \qquad\qquad \text{on } \Omega\subset\mathbb{R}^N,\ \forall t>0\ .\end{aligned}$$

Nondimensionalisation

Analogously to the previous model, let define $\tilde{\boldsymbol{x}} := \frac{\boldsymbol{x}}{W_0}$, $\tilde{t} := \frac{t}{\tau_0}$; this leads to:

$$\begin{aligned} A\partial_{\tilde{t}}\psi &= A^2\tilde{\nabla}^2\psi + \tilde{\nabla}[A^2]\cdot\tilde{\nabla}\psi - \partial_{\tilde{x}}\psi\,\partial_{\tilde{y}}B_{x,y} + \partial_{\tilde{y}}\psi\,\partial_{\tilde{x}}B_{x,y} + \psi(1-\psi^2) - \tfrac{\tilde{\alpha}}{a_2}u(1-\psi^2)^2 \\ \partial_{\tilde{t}}u &= \tilde{\alpha}\tilde{\nabla}^2u + \tfrac{1}{2}\partial_t\psi\ , \qquad \text{on } \tilde{\Omega}\subset\mathbb{R}^N,\ \forall\tilde{t}>0\ , \end{aligned}$$

where $\tilde{\alpha} : \frac{\alpha\tau_0}{W_0^2}$.

Time discretization

The following forward Euler time discretization of the nondimensional anisotropic problem is chosen:

$$\begin{aligned} \Delta\psi^n &= \tfrac{\Delta t}{A^n}\Big\{[A^2]^n\nabla^2\psi^n + \nabla[A^2]^n\cdot\nabla\psi^n - \partial_x\psi^n\,\partial_y[B_{x,y}]^n + \\ &\qquad + \partial_y\psi^n\,\partial_x[B_{x,y}]^n + \psi^n(1-[\psi^n]^2) - \tfrac{\alpha}{a_2}u^n(1-[\psi^n]^2)^2\Big\} \\ \psi^{n+1} &= \psi^n + \Delta\psi^n \\ \partial_t u^{n+1} &= u^n + \alpha\Delta t\,\nabla^2u^n + \tfrac{1}{2}\Delta\psi^n\ , \\ &\qquad\qquad n>0\ , \end{aligned}$$

where the tilde has been omitted for the sake of simplicity, and the following definitions are introduced:

$$\boldsymbol{n}^n = \tfrac{\nabla\psi^n}{|\nabla\psi^n|}\ , \qquad A^n := \begin{cases} A(\boldsymbol{n}^n) & \text{if } |\nabla\psi^n|\neq 0 \\ 1+\varepsilon & \text{otherwise}\ , \end{cases}$$

$$[A^2]^n := [A^n]^2\ , \qquad [B_{x,y}]^n := \begin{cases} B_{x,y}(\boldsymbol{n}^n) & \text{if } |\nabla\psi^n|\neq 0 \\ 0 & \text{otherwise}\ . \end{cases}$$

Space discretization

The same central differences introduced before are used based on the same regular grid. As for the previous model, the numerical scheme for inner nodes then becomes (for $|\nabla\psi^n| \neq 0$):

$$\begin{aligned}
[\delta_x\psi]^n_{i,j} &= \tfrac{\psi^n_{i+1,j}-\psi^n_{i-1,j}}{2h}\\
[\delta_y\psi]^n_{i,j} &= \tfrac{\psi^n_{i,j+1}-\psi^n_{i,j-1}}{2h}\\
[n^2]^n_{i,j} &= ([\delta_x\psi]^n_{i,j})^2 + ([\delta_y\psi]^n_{i,j})^2\\
[n_x]^n_{i,j} &= \tfrac{[\delta_x\psi]^n_{i,j}}{[n^2]^n_{i,j}}\\
[n_y]^n_{i,j} &= \tfrac{[\delta_y\psi]^n_{i,j}}{[n^2]^n_{i,j}}\\
a^n_{i,j} &= A(\varphi^n_{i,j})\\
[\delta_{n_x}a]^n_{i,j} &= \partial_{n_x}A([n_x]^n_{i,j},[n_y]^n_{i,j})\\
[\delta_{n_y}a]^n_{i,j} &= \partial_{n_y}A([n_x]^n_{i,j},[n_y]^n_{i,j})\\
[b_{x,y}]^n_{i,j} &= \tfrac{a^n_{i,j}}{([n^2]^n_{i,j})^{1/2}}([\delta_x\psi]^n_{i,j}[\delta_{n_y}a]^n_{i,j} - [\delta_y\psi]^n_{i,j}[\delta_{n_x}a]^n_{i,j})\\
\Delta\psi^n_{i,j} &= \tfrac{\Delta t}{a^n_{i,j}}\Big\{[a^n_{i,j}]^2\tfrac{\psi^n_{i+1,j}+\psi^n_{i-1,j}+\psi^n_{i,j+1}+\psi^n_{i,j-1}-4\psi^n_{i,j}}{h^2} +\\
&\quad + \left[\tfrac{[a^n_{i+1,j}]^2-[a^n_{i-1,j}]^2}{2h} - \tfrac{[b_{x,y}]^n_{i,j+1}-[b_{x,y}]^n_{i,j-1}}{2h}\right][\delta_x\psi]^n_{i,j} +\\
&\quad + \left[\tfrac{[b_{x,y}]^n_{i+1,j}-[b_{x,y}]^n_{i-1,j}}{2h} + \tfrac{[a^n_{i,j+1}]^2-[a^n_{i,j-1}]^2}{2h}\right][\delta_y\psi]^n_{i,j} +\\
&\quad + \psi^n_{i,j}(1-[\psi^n_{i,j}]^2) - \tfrac{\alpha}{a_2}u^n_{i,j}(1-[\psi^n_{i,j}]^2)^2\Big\}\\
\psi^{n+1}_{i,j} &= \psi^n_{i,j} + \Delta\psi^n_{i,j}\\
u^{n+1}_{i,j} &= u^n_{i,j} + \alpha\Delta t\nabla^2 u^n_{i,j} + \tfrac{1}{2}\Delta\psi^n_{i,j}
\end{aligned} \tag{4.2}$$

$$0 < i < N_x,\ 0 < j < N_y,\ n > 0\ .$$

4.1.3 Interface Characterization

As phase field models assume a quantitative meaning only in the sharp-interface limit (2.8), it is important to compute the variables appearing therein starting from the numerical phase field solution. These quantities are the interface position, x, velocity, v, and curvature, κ. Their computation is strongly dependent on the accuracy of the interface position evaluation defined as the distance from the origin of the 0-level of the phase field. Assuming a dendrite arm growing in the direction of the positive x semi axis, and that indexing is chosen such that $y_{\bar{j}} = 0$, the following Algorithm 1 has been implemented as in [49].

Algorithm 1

```
1: for all timestep $k$ do
2:   identify the minimum $\bar{\imath}$ such that $x_{\bar{\imath}} > 0$ and that $\psi_{\bar{\imath}-1,\bar{\jmath}}\psi_{\bar{\imath}+1,\bar{\jmath}} < 0$
3:   compute the fourth-order polynomial interpolant, $\tilde{\psi}^n(x)$, of $\psi^n$ at nodes $x_i$ for $i = \bar{\imath}-2,\dots,\bar{\imath}+2$
4:   compute the interface position $x^n_{\text{intf}}$ as the root of $\tilde{\psi}^n(x)$
5:   compute tip velocity as $v^n_{\text{intf}} = \frac{x^n_{\text{intf}} - x^{n-1}_{\text{intf}}}{\Delta t}$
6:   for $i = \bar{\imath}-2$ to $\bar{\imath}+2$ do
7:     compute the fourth-order polynomial interpolant, $\tilde{\psi}^n_i(y)$, of $\psi^n$ at nodes $y_j$ with $j = \bar{\jmath}-2,\dots,\bar{\jmath}+2$;
8:     compute $\delta^2_y\tilde{\psi}^n_i = \partial^2_y\tilde{\psi}^n_i(0)$
9:   end for
10:  compute $\delta^2_y\tilde{\psi}^n(x)$ as the fourth-order polynomial interpolant of $\delta^2_y\tilde{\psi}^n_i$ at points $x_i$ with $i = \bar{\imath}-2,\dots,\bar{\imath}+2$
11:  for $j = \bar{\jmath}-2$ to $\bar{\jmath}+2$ do
12:    compute the fourth-order polynomial interpolant, $\tilde{\psi}^n_j(x)$, of $\psi^n$ at nodes $x_i$ with $i = \bar{\imath}-2,\dots,\bar{\imath}+2$;
13:    compute $\delta_x\tilde{\psi}^n_j = \partial_x\tilde{\psi}^n_j(x_{\text{intf}})$
14:  end for
15:  compute $\delta_x\tilde{\psi}^n(y)$ as the fourth-order polynomial interpolant of $\delta_x\tilde{\psi}^n_j$ at points $y_j$ with $j = \bar{\jmath}-2,\dots,\bar{\jmath}+2$
16:  compute $\kappa^n_{\text{intf}} = [\rho^n_{\text{intf}}]^{-1} \approx \frac{\delta^2_y\tilde{\psi}^n(x_{\text{intf}})}{|\delta_y\tilde{\psi}^n(0)|}$
17: end for
```

A simpler algorithm for calculating the tip properties is the following:

Algorithm 2

```
1: for all timestep $k$ do
2:   identify the minimum $\bar{\imath}$ such that $x_{\bar{\imath}} > 0$ and that $\psi_{\bar{\imath}-1,\bar{\jmath}}\psi_{\bar{\imath}+1,\bar{\jmath}} < 0$
3:   for $j = \bar{\jmath}-2$ to $\bar{\jmath}+2$ do
4:     compute the fourth-order polynomial interpolant, $\tilde{\psi}^n_j(x)$, of $\psi^n$ at nodes $x_i$ for $i = \bar{\imath}-2,\dots,\bar{\imath}+2$
5:     compute the interface positions $x^n_{0,j}$ as the root of $\tilde{\psi}^n_j(x)$
6:   end for
7:   set the tip position $x^n_{\text{intf}} = x^n_{0,0}$
8:   compute tip velocity as $v^n_{\text{intf}} = \frac{x^n_{\text{intf}} - x^{n-1}_{\text{intf}}}{\Delta t}$
9:   compute the fourth-order polynomial interpolant of the interface, $\tilde{x}^n_0(y)$, of $x^n_{0,j}$ at nodes $y_j$
10:  compute $\kappa^n_{\text{intf}} = [\rho^n_{\text{intf}}]^{-1} \approx \partial^2_y\tilde{x}^n_0(0)$
11: end for
```

In literature [49, 77, 90], a parabolic model for the dendrite tip shape is often adopted for the anisotropic case. For comparison purpose, the parabolic curvature and radius of curvature can be computed from the parabolic interpolant of the contour level corresponding to $\psi = 0$ for a tip growing along one of the directions favoured by anisotropy. In doing this it is important to omit points close to the actual arm tip (see Figure 4.1), since the actual shape of the interface is deviating from being parabolic close to the tip. It is important to remark that the definition of such a radius is arbitrary being it strongly dependent on the choice of interpolation points and method.

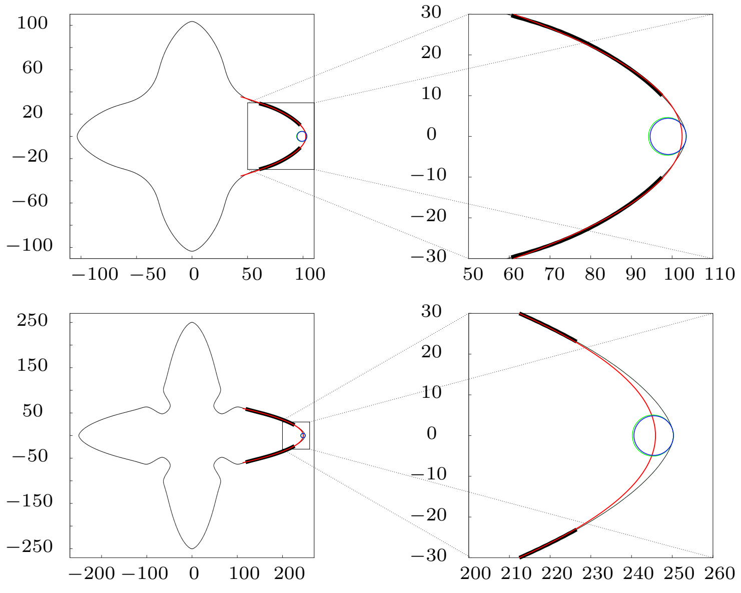


Figure 4.1: Tip radius calculation at two different time, $t = 1000$ (top), and $t = 3000$ (bottom): the local osculating circles, computed with Algorithm 1 (blue) and with Algorithm 2 (green), are compared with the interpolant parabola (red) computed using the portion of the interface contour represented by the thick black line. In the zoomed plots on the right, it is clear how the local curvature gives an accurate description of the interface in the neighbour of the tip, while the parabolic fit of the arm is a good model for the dendrite arm far from the tip. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $M = 400$, $h = 0.4$, $\Delta t = 0.015$]

4.1.4 Validation

The first simulations here reported have the purpose of validate the results of this first serial implementation in two dimensions. To assess the quality of the software, the interface evolution over time, has been compared, at first, qualitatively with the results reported in [49], and then, quantitatively in terms of the interface tip velocity, and curvature values obtained by post-processing the computed output and in particular their steady-state values.

Interface Evolution

Thanks to the symmetry of the isotropic model, it is possible to reduce the computational time needed for simulations limiting the domain on which the solutions are actually computed. In particular, choosing $\Omega = [0, M]^2$ and imposing Neumann boundary conditions on $\Gamma_N := (\{0\} \times [0, M] \cup [0, M] \times \{0\}$, it is possible to use a grid which is four times smaller than the one corresponding to the same level of accuracy on the full domain.

In order to asses the reliability of the implemented solution for the anisotropic model, the evolution of liquid-solid interface over time has been considered. Results have been compared

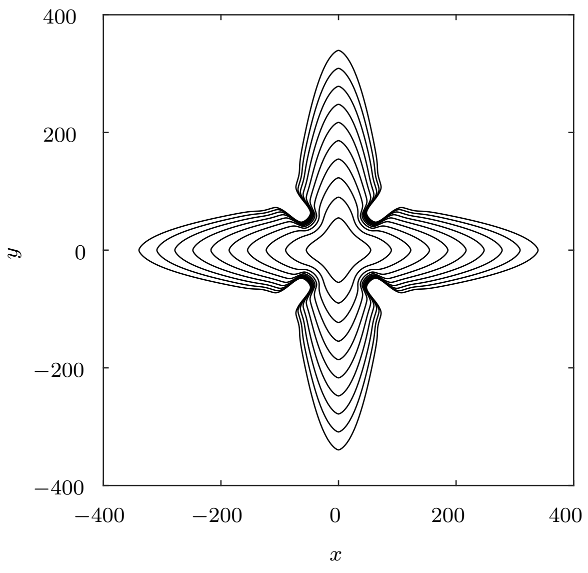


Figure 4.2: Sequence of interface pattern shown every 25000 iterations. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $M = 1400$, $h = 0.4$, $\Delta t = 0.015$]

with those available in literature [79, 48, 49] and good qualitative accordance has been found. A typical sequence of dendritic shapes is shown in Figure 4.2, where the 0-contour level of the phase field is plotted every 25000 iterations. The evolution of the temperature field over time is shown in Figure 4.3, as a sequence of colour plots.

Two numerical phenomena could arise and affect the reliability of the simulations. The first one is related to the space discretization step, h, and the thermal diffusivity α and is known as *grid anisotropy*: due to the low order finite difference approximation, diffusion is observed to be favoured along the bisectors of the I-III and II-IV quadrants. This numerical phenomenon is empathized by low values of thermal diffusivity and appears to vanish when $h \to 0$.

The second numerical issue is a consequence of the explicit time discretization. The computed solution is observed to oscillate around the physical one with increasing magnitude when the time discretization step is too long. In particular, the explicit time discretization scheme introduces a numerical stability constraint on the timestep [76]:

$$\Delta t < Ch^2 \, , \tag{4.3}$$

where C is a suitable constant depending on the model parameters and initial condition.

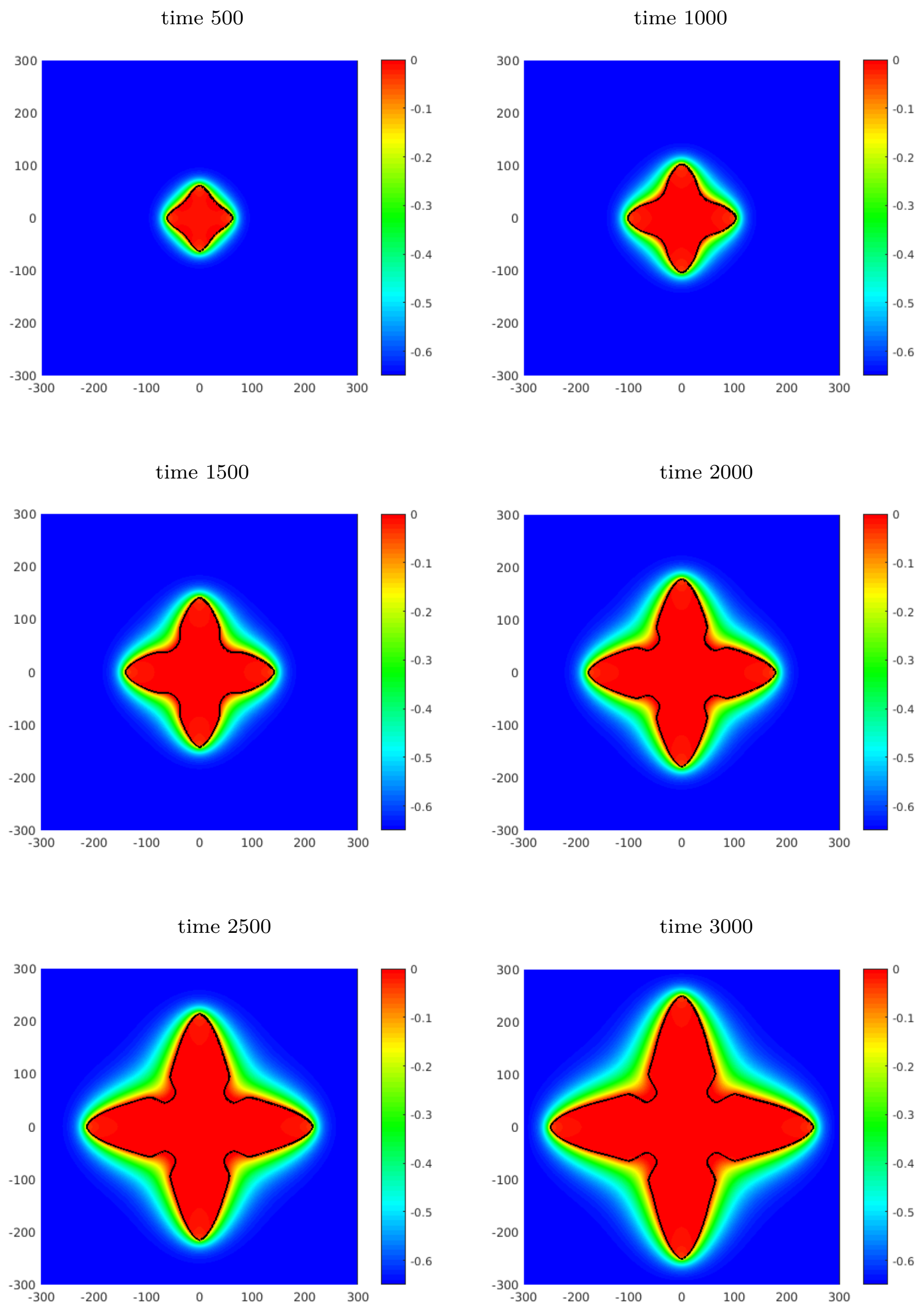


Figure 4.3: Nondimensional temperature field evolution at time intervals of length 500. The 0-contour of ψ (black) corresponds to the liquid/solid interface. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $M = 300$, $h = 1$, $\Delta t = 0.1$]

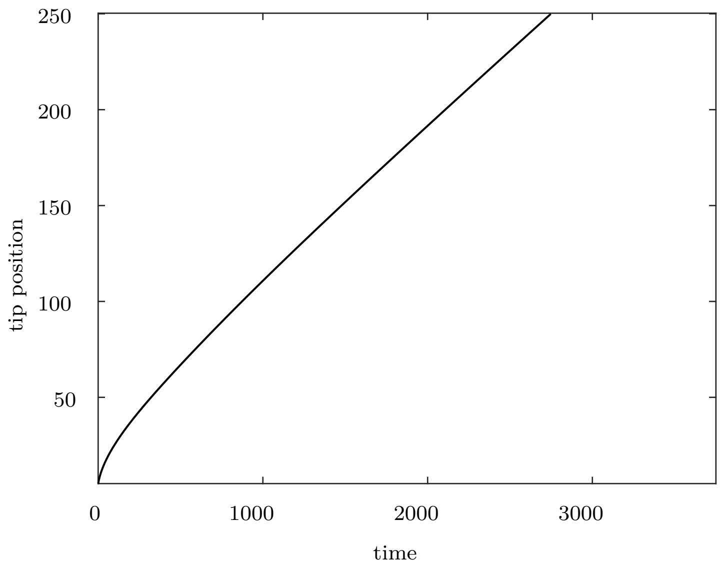


Figure 4.4: Tip position profile. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $M = 1400$, $h = 0.4$, $\Delta t = 0.015$]

Interface Characterization

In order to asses quantitatively the reliability of the results produced by this implementation, the steady-state tip velocity is compared with that computed by the Green function method as reported in Table II by Karma et al. [49],

$$V_{\text{tip}} = \frac{D\tilde{V}_{\text{tip}}}{d_0} \, ,$$

where $\tilde{V}_{\text{tip}}$ is the dimensionless tip velocity as defined therein.

In Figures 4.4–4.6 are report the tip position, velocity and curvature profiles computed as described in Algorithm 1 and in Algorithm 2.

In Figure 4.5 it has been plotted a reference line in correspondence to the steady-state velocity calculated by the Green function method whose value for this choice of parameters is 0.847. This graph shows that there is also a good quantitative comparison between this implementation and literature results.

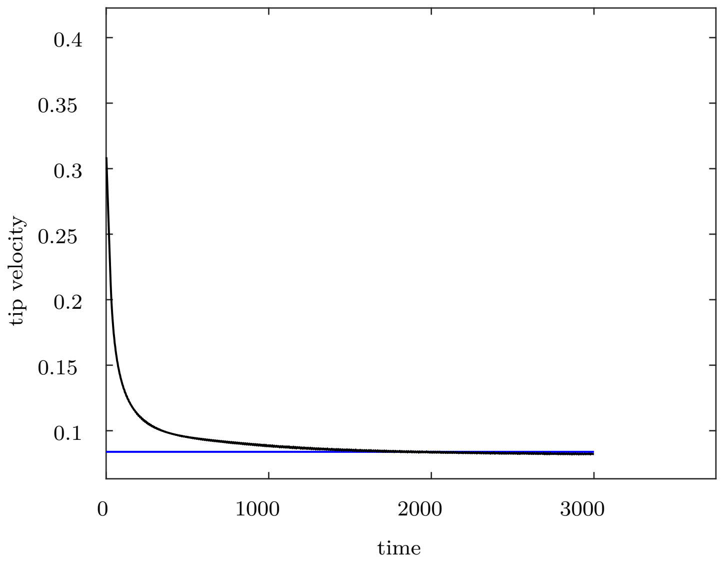


Figure 4.5: Tip velocity profile. In blue, the reference steady-state velocity value computed by the Green function method. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $M = 1400$, $h = 0.4$, $\Delta t = 0.015$]

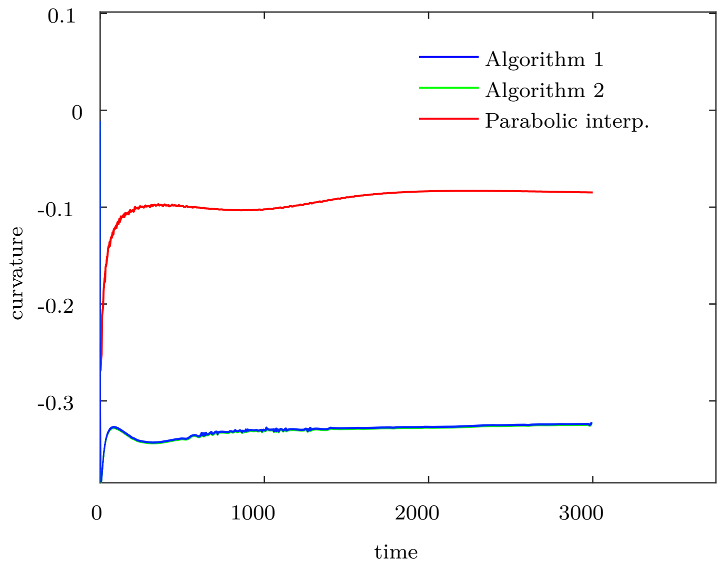


Figure 4.6: Curvature profiles. Comparison between local tip curvature computed with Algorithm 1 (blue) and in Algorithm 2 (green) and the parabolic arm curvature (red). [$\alpha = 1$, $\varepsilon = 0.05$, $M = 1400$, $h = 0.4$, $R_0 = 5$, $\Delta = 0.65$, $\Delta t = 0.015$]

4.2 PhaseField Uintah Component

In the second phase of the development, focus has been given to the implementation of the same schemes presented in §4.1 within the Uintah Computational Framework. This second phase was articulated in three steps. In the first step (§4.2.1) adaptive mesh refinement techniques are not introduced in the new component aiming to achieve parallelization on its own. This step is validated and its parallel performance assessed in §4.2.2.

Then, in §4.2.3, adaptivity is added to the component and, in §4.2.4, results obtained with new implementation have been compared with those computed on uniform grids with the first implementation.

The third step of this phase of development (§4.2.5) has been the implementation of the phase field model (2.5) in three dimension.

4.2.1 Parallelisation – Implementation

As described in §2.2, within Uintah parallelism is achieve by splitting, both, the simulation into tasks and the computational domain into patches which are a partition of the grid among processing units. In order to make it possible for the Uintah TaskGraph Compiler to build a task-graph, it is essential to specify correctly dependencies, width and kind of the ghost cells which a task relies on, and which variables are created, computed, or updated within that task.

In writing the component for scheme (4.2), the computation of the solution at each timestep has been split into three tasks:

Task 1 Compute the gradient of the phase field and its norm using the previous solution.

$$\begin{aligned}
[\delta_x \psi]^n_{i,j} &= \tfrac{\psi^n_{i+1,j} - \psi^n_{i-1,j}}{2h} \\
[\delta_y \psi]^n_{i,j} &= \tfrac{\psi^n_{i,j+1} - \psi^n_{i,j-1}}{2h} \\
[n^2]^n_{i,j} &= ([\delta_x \psi]^n_{i,j})^2 + ([\delta_y \psi]^n_{i,j})^2 \ .
\end{aligned}$$

This task depends only on the value of the phase field at the previous time step $\boldsymbol{\psi}^n$ and computes the derivative of the phase field along the coordinate directions, $\boldsymbol{\delta_x \psi}^n$ and $\boldsymbol{\delta_y \psi}^n$ and the square of its norm, $[\boldsymbol{n^2}]^n$. Since for each index (i, j) this formula requires that the values of the phase field is known at adjacent nodes/cells, it has to be specified to the framework that this task requires one ghost node/cell around each side of the patch.

Task 2 Compute the terms depending on the anisotropy function and its derivatives. When using the 2D formulation (2.7) for the anisotropy function the relevant equation from the space discretization (4.2) becomes:

if $[n^2]^n_{i,j} = 0$,
$$\begin{aligned} a^n_{i,j} &= 1+\varepsilon \\ [a^2]^n_{i,j} &= (a^n_{i,j})^2 \\ [b_{x,y}]^n_{i,j} &= 0\ ; \end{aligned}$$

otherwise ,
$$\begin{aligned} a^n_{i,j} &= 1+\varepsilon\{4\tfrac{([\delta_x\psi]^n_{i,j})^4+([\delta_y\psi]^n_{i,j})^4}{([n^2]^n_{i,j})^2}-3\} \\ [a^2]^n_{i,j} &= (a^n_{i,j})^2 \\ [b_{x,y}]^n_{i,j} &= 16\varepsilon a^n_{i,j}\tfrac{[\delta_x\psi]^n_{i,j}[\delta_y\psi]^n_{i,j}}{[n^2]^n_{i,j}}\tfrac{([\delta_x\psi]^n_{i,j})^2-([\delta_y\psi]^n_{i,j})^2}{[n^2]^n_{i,j}}\ . \end{aligned}$$

This task depends on the value of the component and the square of the norm of the gradient of the phase field, $\boldsymbol{\delta_x\psi}^n$, $\boldsymbol{\delta_y\psi}^n$ and $[\boldsymbol{n^2}]^n$, and computes the quantities $\boldsymbol{a}^n$, $[\boldsymbol{a^2}]^n$, $[\boldsymbol{b_x}]^n$, and $[\boldsymbol{b_y}]^n$. It does not require information exchange between patches.

Task 3 Compute the new phase and thermal fields at next time-step.

$$\begin{aligned} \Delta\psi^n_{i,j} &= \tfrac{\Delta t}{a^n_{i,j}}\Big\{[a^n_{i,j}]^2\tfrac{\psi^n_{i+1,j}+\psi^n_{i-1,j}+\psi^n_{i,j+1}+\psi^n_{i,j-1}-4\psi^n_{i,j}}{h^2}+ \\ &+\left(\tfrac{[a^n_{i+1,j}]^2-[a^n_{i-1,j}]^2}{2h}-\tfrac{[b_{x,y}]^n_{i,j+1}-[b_{x,y}]^n_{i,j-1}}{2h}\right)[\delta_x\psi]^n_{i,j}+ \\ &+\left(\tfrac{[b_{x,y}]^n_{i+1,j}-[b_{x,y}]^n_{i-1,j}}{2h}+\tfrac{[a^n_{i,j+1}]^2-[a^n_{i,j-1}]^2}{2h}\right)[\delta_y\psi]^n_{i,j}+ \\ &+\psi^n_{i,j}(1-[\psi^n_{i,j}]^2)-\tfrac{\alpha}{a_2}u^n_{i,j}(1-[\psi^n_{i,j}]^2)^2\Big\} \\ \psi^{n+1}_{i,j} &= \psi^n_{i,j}+\Delta\psi^n_{i,j} \\ u^{n+1}_{i,j} &= u^n_{i,j}+\alpha\Delta t\nabla^2u^n_{i,j}+\tfrac{1}{2}\Delta\psi^n_{i,j}\ . \end{aligned}$$

This task depends on both the values of the solution at previous time step, $\boldsymbol{\psi}^n$, $\boldsymbol{u}^n$ and the variables computed by *Task 1* ($\boldsymbol{\delta_x\psi}^n$, $\boldsymbol{\delta_y\psi}^n$ and $[\boldsymbol{n^2}]^n$), and *Task 2* ($\boldsymbol{a}^n$, $[\boldsymbol{a^2}]^n$, $[\boldsymbol{b_x}]^n$, and $[\boldsymbol{b_y}]^n$), and requires a one node/cell layer of ghosts around patches for $\boldsymbol{\psi}^n$, $\boldsymbol{u}^n$, $[\boldsymbol{a^2}]^n$, $[\boldsymbol{b_x}]^n$, and $[\boldsymbol{b_y}]^n$.

The reason for splitting the computation of the new solution over tree tasks and not merging the first two is that the computation of the phase field gradient is a task that may be performed for estimating the error, and therefore deciding where mesh refinement should be performed. Even though the current implementation does not use error estimation adaptivity yet (see §4.2.3), this separation will be useful in further improvements.

4.2.2 Parallelisation – Results

For this implementation, as for the serial one, the first result presented is the validation of the software. Simulations are analysed, qualitatively, plotting the interface evolution over time and, quantitatively, comparing the interface tip velocity, and curvature values obtained by post-processing the computed output to the previous results. Eventually, some scalability tests have been performed to asses the level of parallelism achieved.

Interface Evolution

To asses the quality of the solution computed by the component within Uintah, evolution of the interface and of the temperature field has been compared to results computed using the previous implementation. In Figure 4.7 the results computed using the cell-centred and the node-centred variables' representation are shown at successive timesteps. As a second quality assessment, the same algorithms described in §4.1 are used to compute the tip position, velocity and curvatures profiles which, then, have been compared with the respective ones computed with the previous implementation. The maximum relative error between such profiles does not exceed 2% making a graphical comparison of them not appreciable.

Parallel Scalability

To analyse the properties of the parallel implementation within Uintah a series of simulations has been performed using an increasing number of computational nodes. All simulations have been performed on ARC3, the HPC service of the University of Leeds. Its standard nodes have 24 cores at 2.2GHz and 128GB of memory.

Both weak and strong scalability tests have been considered. Weak scalability tests measure how a software performs when the number of cores is increased while keeping the working load constant, i.e. the problem size per processor is kept constant. It is a useful tool, for example, to understand the parallel performance of a software when resources are fully exploited.

On the contrary strong scalability tests are used to measure the performance of a software when the number of cores is increased while keeping the global problem size constant. These tests are a useful tool to quantify what fraction of the whole calculation is parallelized: in fact, only if the computation is fully parallelized, an inverse proportionality is observed between computation time and number of processors.

Tests have been performed using 1, 4, 16 and 64 nodes corresponding to 24, 96, 384 and 1536 cores. It has been tested that using 864 million cells per node corresponds to using more than the 96% of the memory available, when cell centred data representation is used, while for the node centred case 384 million cells per node are enough to fill more than 90% of the memory. The larger use of memory of the latter is to be addressed to the fact that for each cell in Uintah is three dimensional even for two dimensional simulations, roughly doubling the size of variables.

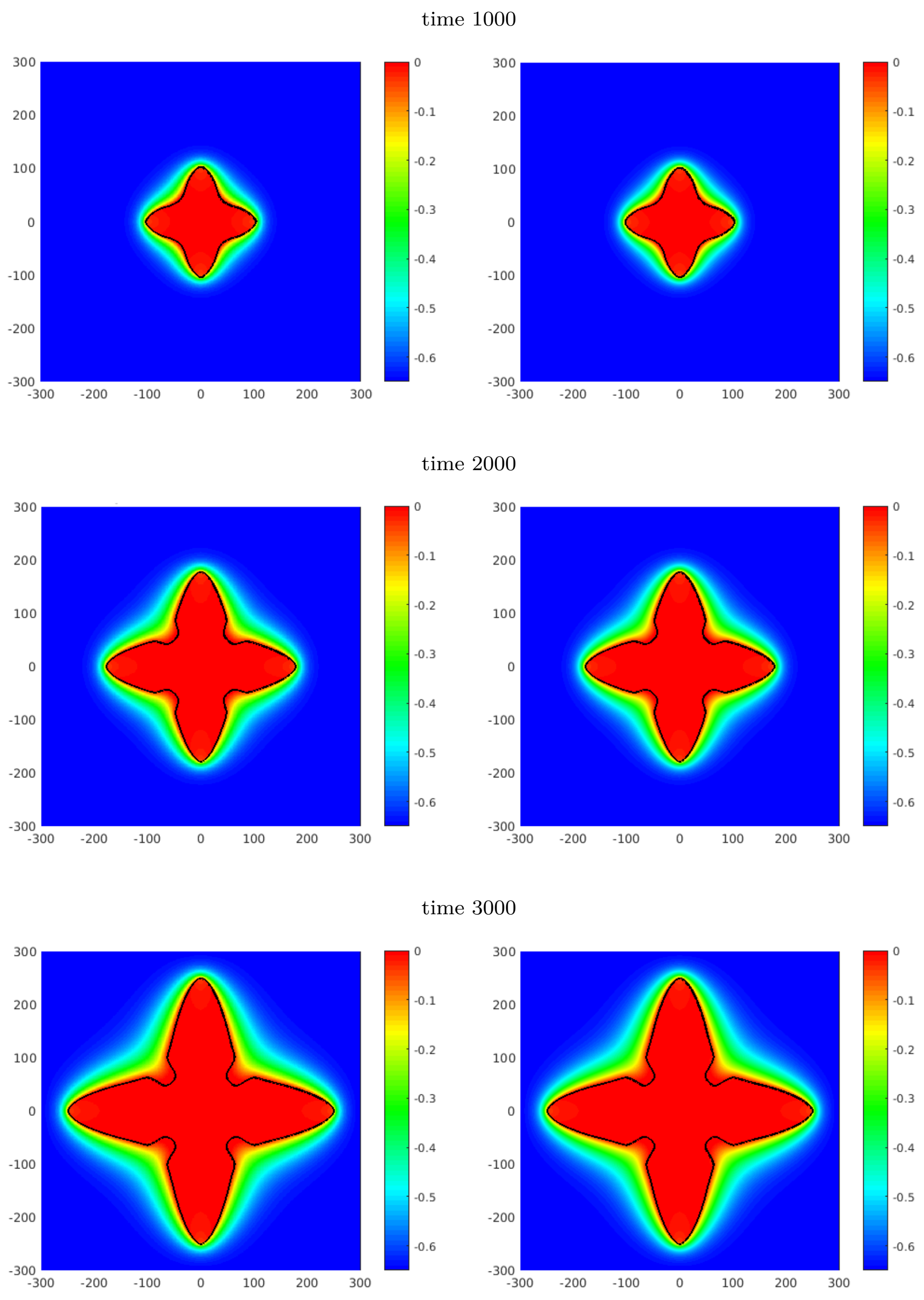


Figure 4.7: Comparison between the nondimensional temperature field computed using cell-centred (left) and node-centred (right) variables at time intervals of length 1000. The 0-contour of ψ (black) corresponds to the liquid/solid interface. $[\alpha = 1,\ \varepsilon = 0.05,\ R_0 = 5,\ \Delta = 0.65,\ \Omega = [-300, 300]^2,\ h = 1,\ \Delta t = 0.1]$

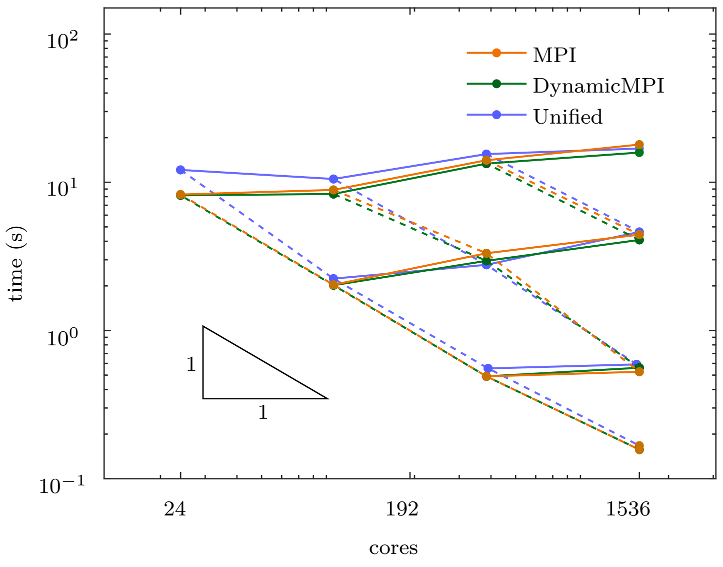


Figure 4.8: Scalability for the cell centred case. Results for different choices of the scheduled are shown. Week scalability paths are drawn with solid lines, the top-most of which corresponds to a almost complete use the memory resources. Strong scalability paths (dashed) can be compared to the reference triangle corresponding to ideal strong scalability.

In Figures 4.8 and 4.9 the execution time of the simulations performed is reported as a function on the number of cores utilized for the cell centred and the node centred cases respectively.

In general the node centred implementation seems to have slightly better weak scalability performance especially when the use of memory is more abundant. Both implementation show, overall, almost optimal scaling behaviours, either weak and strong, within the range tested.

4.2.3 Adaptive Mesh Refinement – Implementation

The application of Adaptive Mesh Refinement methods to the previous parallel component is achieved within the Uintah Computational Framework by telling to the task-graph compiler which task to perform for each of the following activities:

Refine tasks belonging to this activity are executed every time that, after the initial time-step, it is required to refine a given level of the mesh. These tasks are responsible for defining the solution at the new finest level, usually by interpolation of the solution at coarser levels. Such tasks are not executed when refinement happens on the initial time-step, since in this case the initialization tasks are executed instead.

Coarsen tasks belonging to this activity are executed after every time-step. They are responsible to copy the solution from the finest to the coarsest level, by restriction or average of

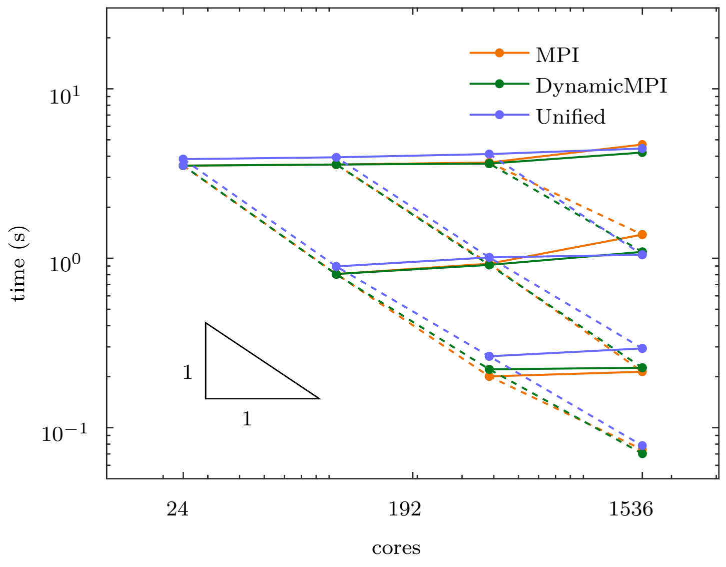


Figure 4.9: Scalability for the node centred case. Results for different choices of the scheduled are shown. Week scalability paths are drawn with solid lines, the topmost of which corresponds to a almost complete use the memory resources. Strong scalability paths (dashed) can be compared to the reference triangle corresponding to ideal strong scalability.

the solution at the finer level.

InitialErrorEstimate tasks belonging to this activity are executed before the first time-step. The aim of these tasks is to set a refinement flag on every patch and level in order to control whether refinement or coarsening is there required.

ErrorEstimate tasks belonging to this activity are executed before every time-step but the first. The aim of these tasks is the same of those belonging to the *InitialErrorEstimate* activity.

For the sake of simplicity, in current implementation, the assumption that boundary edges of every level are far enough from the region where the solidification process occurs has been made. To make this assumption hold, the same task has been used for both *InitialErrorEstimate* and *ErrorEstimate* activities. This task sets the refinement flag to true on every patch on which $[\boldsymbol{n}^2] < \texttt{tol}$, for a given small positive tolerance value, `tol`. Thanks to this assumption, at this stage, it has been possible to avoid the imposition of any boundary conditions to grant the continuity of the solution between levels. This assumption is not optimal since does not spread the approximation error on the solution evenly over the grid in order to achieve a desired level of accuracy, and will become restrictive for more complex phase field models. For this reasons, it is planned to implement inter-level boundary condition in the next months.

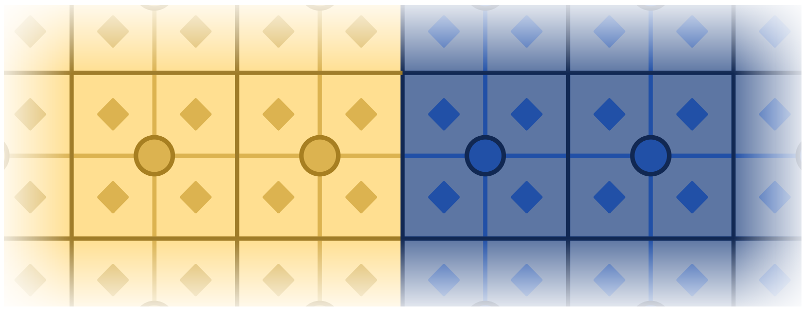


Figure 4.10: Grid refinement with cell centred data. Variables' values on the new grid level (diamonds) are copied from the values at corresponding cell on the coarser level (circles). No data exchange is necessary between adjacent patches.

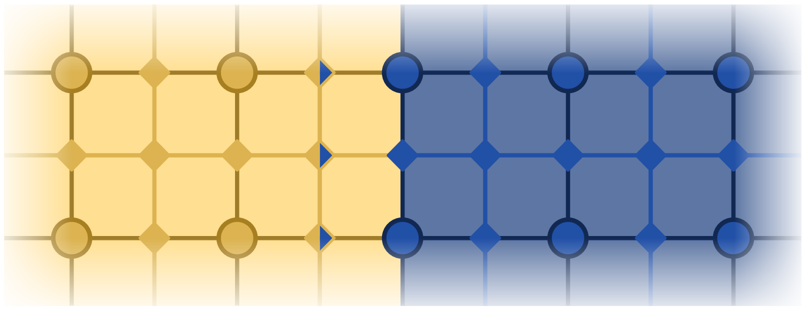


Figure 4.11: Grid refinement with node centred data. Variables' values on the new grid level corresponding to new nodes (diamonds) are interpolated from the values at neighbour nodes on the coarser level (circles). When the new node lies on a cell that shares an edge/face with a cell on an adjacent patch, exchange of information may be required among those patches.

In current implementation one task for each of the previous activities has been introduced:

Refine the implementation of the refinement task is dependent on the representation of the variables as node or cell centred. In the former case it is possible to copy the variables' nodal values from the coarser to the finer level and compute the remaining nodal values by interpolation. In this case (see Figure 4.11) it is not possible to state that all nodes needed for the interpolation process belong to the same patch and it is needed to specify its dependency on ghost nodes. On the contrary, when we are refining cell centred variables, no interpolation is needed and it is possible to simply copy a value from a coarser cell to its corresponding finer cells (see Figure 4.10) without using ghost cells.

Coarsen the implementation of this task, as well as the *Refine* one, is dependent on the variables' representation. In the node-centred case, all nodes in the coarser level are also on the finer one and thus it is sufficient to copy values from the latter to the first. Conversely, when variables representation is cell-centred, the value on the coarser level is computed as the average of the values at the corresponding cells on the finer level. In both cases it is not necessary to use ghost elements.

InitialErrorEstimate/ErrorEstimate this task must set the refinement flag on a cell basis independently of the phase field representation. When node-centred representation is used, the norm of the gradient at a given cell is estimated with first order finite difference using the vertices of that cell while in the cell-centred case, finite differences are applied on surrounding cells. In both cases it is necessary to use one layer of ghost elements.

4.2.4 Results

For this implementation the same validation and weak scalability tests as for the previous implementation within Uintah not involving AMR have been performed.

Interface Evolution

The first comparison performed between the implementation with Adaptive Mesh Refinement techniques and those not involving them, is qualitative. In Figure 4.12 colour plots of the nondimensional temperature field together with the curve $\psi = 0$ which corresponds to the liquid/solid interface are reported for tree different timesteps. Those simulations have been performed using three grid levels, the coarsest of which with $h = 4$, and a refinement ratio of 2, so that the finest level has the same grid spacing of the simulations performed without AMR.

Two different phenomena affect these results: when using adaptivity together with cell centred data representation, they are observed a speed up of the dendrite growth, and lower values of curvature for the dendrite tips and arms. When, instead, adaptivity is used in conjunction with node-centred variables no such speed up is find but the temperature field diffuse more at coarser levels.

In Figures 4.13–4.15 it is possible to appreciate quantitatively how such phenomena affect the dendrite arm properties.

It is still to study the dependency of the aforementioned behaviours with respect to discretization and adaptivity parameters, but it is plausible that they have been amplified by the very large space discretization step on the coarsest level.

4.2.5 3D Model – Implementation

In this third step, the three dimensional model for pure metal solidification is developed. The component described in §4.2.1 is extended to this case redefining the task introduced therein keeping the same structure for the component.

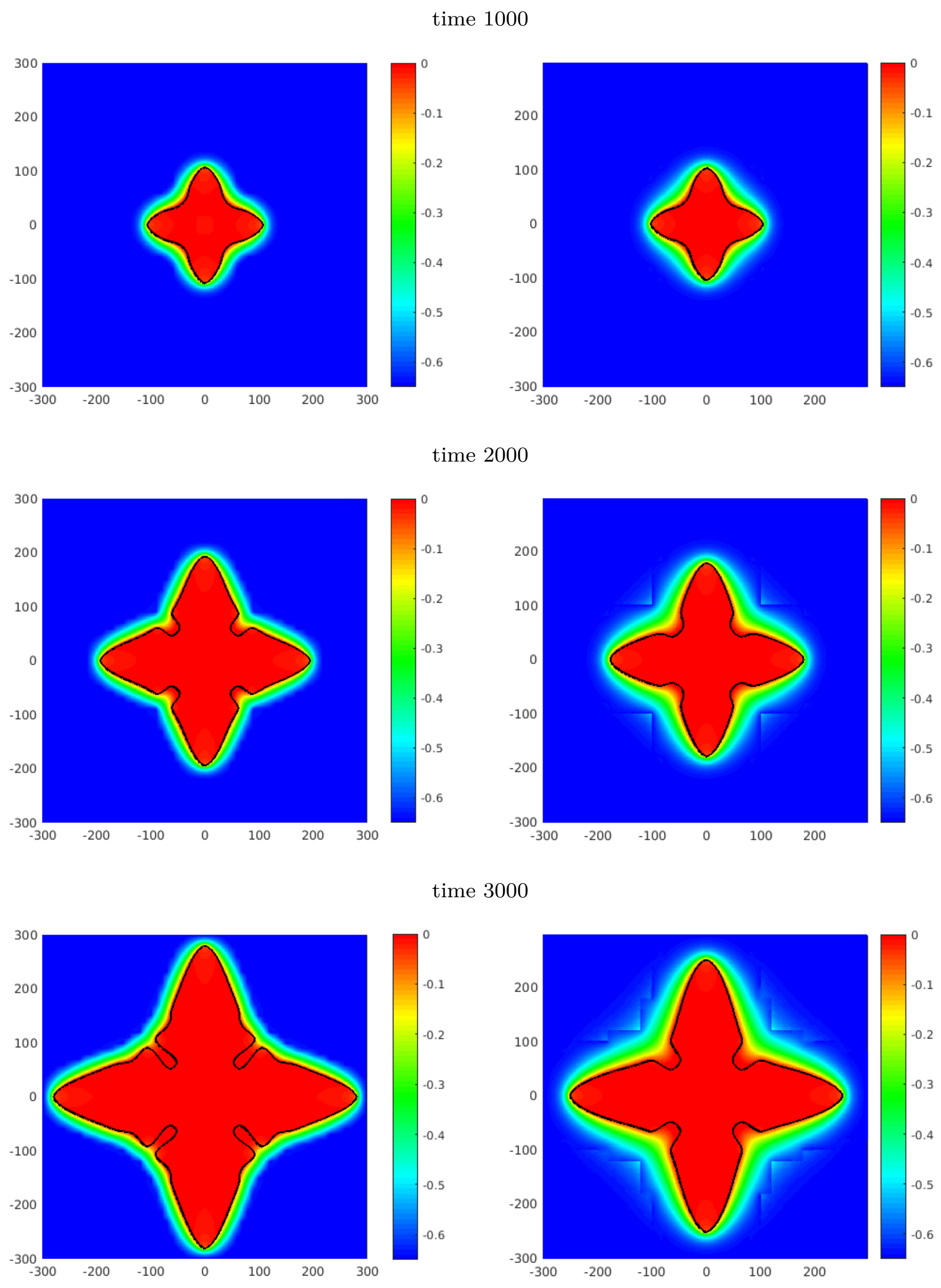


Figure 4.12: Comparison between the nondimensional temperature field computed using AMR and cell-centred (left) and node-centred (right) variables at time intervals of length 1000. The 0-contour of ψ (black) corresponds to the liquid/solid interface. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $\Omega = [-300, 300]^2$, $h = \{4, 2, 1\}$, $\Delta t = 0.1$]

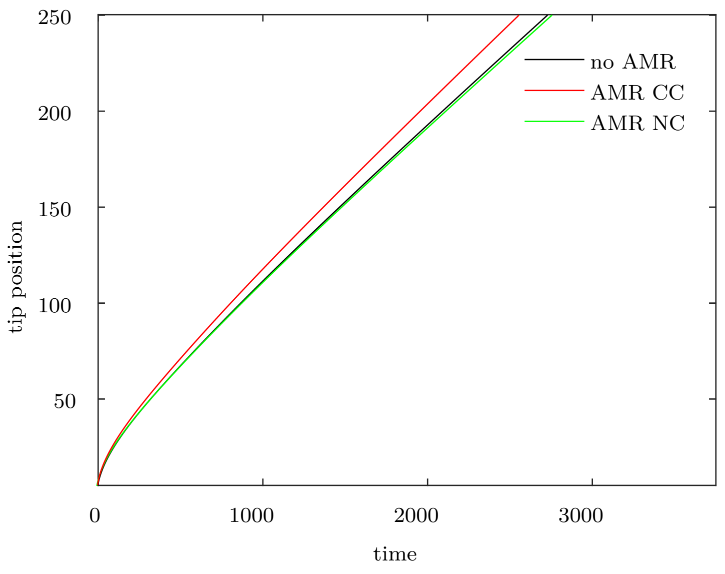


Figure 4.13: Tip position profile. [$\alpha = 1$, $\varepsilon = 0.05$, $\Omega = [-560, 560]^2$, $h = \{3.2, 1.6, 0.8, 0.4\}$, $R_0 = 5$, $\Delta = 0.65$, $\Delta t = 0.015$]

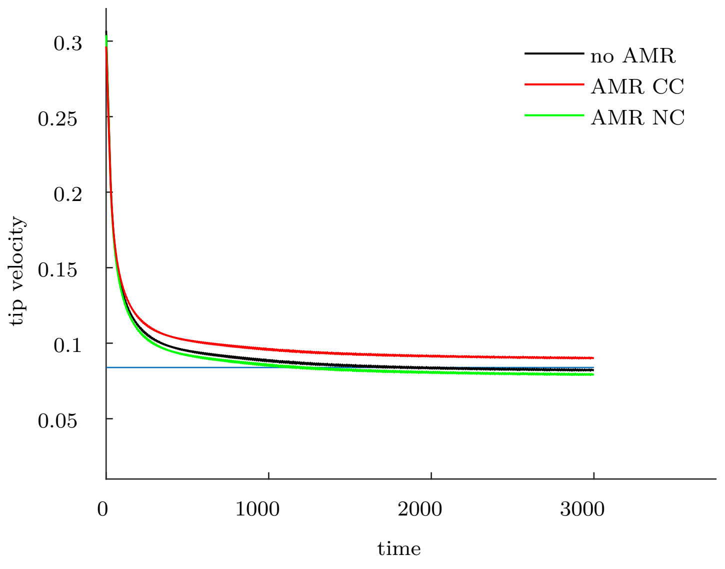


Figure 4.14: Tip velocity profile. In blue, the reference steady-state velocity value computed by the Green function method. [$\alpha = 1$, $\varepsilon = 0.05$, $\Omega = [-560, 560]^2$, $h = \{3.2, 1.6, 0.8, 0.4\}$, $R_0 = 5$, $\Delta = 0.65$, $\Delta t = 0.015$]

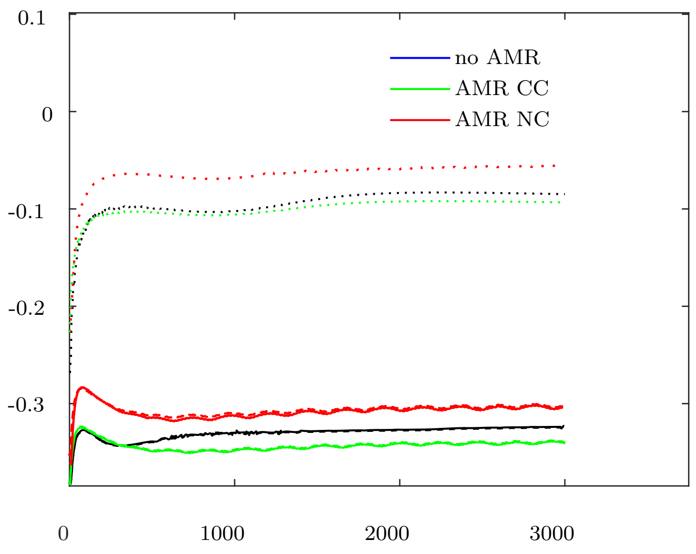


Figure 4.15: Curvature profiles. Local tip curvature computed with Algorithm 1 (blue) and in Algorithm 2 (green) is compare with the parabolic arm curvature (red). [$\alpha = 1$, $\varepsilon = 0.05$, $\Omega = [-560, 560]^2$, $h = 1.6, 0.8, 0.4$, $R_0 = 5$, $\Delta = 0.65$, $\Delta t = 0.015$]

3D Anisotropic Model

With the usual choice of h and f model (2.3) for the three dimensional anisotropic case can be rewritten, following the same procedure as in §4.2.1, as follows:

$$\begin{aligned}
\tau_0 A \partial_t \psi = {} & W_0^2 A^2 \Delta\psi + \nabla[A^2]\cdot\nabla\psi - W_0^2\, \partial_x\psi\, \partial_y B_{x,y} - W_0^2\, \partial_x\psi\, \partial_z B_{x,z} + \\
& - W_0^2\, \partial_y\psi\, \partial_x B_{x,y} + W_0^2\, \partial_y\psi\, \partial_z B_{y,z} + W_0^2\, \partial_z\psi\, \partial_z B_{x,z} + W_0^2\, \partial_z\psi\, \partial_y B_{y,z} + \\
& + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\
\partial_t u = {} & \alpha\Delta u + \tfrac{1}{2}\partial_t\psi\ , \qquad \text{on } \Omega,\ \forall t > 0\ .
\end{aligned}$$

Nondimensionalization Defining, as for the other models, $\tilde{\boldsymbol{x}} := \frac{\boldsymbol{x}}{W_0}$, $\tilde{t} := \frac{t}{\tau_0}$, the following nondimensional model is derived:

$$\begin{aligned}
A \partial_{\tilde t}\psi = {} & A^2 \tilde\nabla^2\psi + \tilde\nabla[A^2]\cdot\tilde\nabla\psi + \\
& - W_0^2\, \partial_{\tilde x}\psi\, \partial_{\tilde y} B_{x,y} - W_0^2\, \partial_{\tilde x}\psi\, \partial_{\tilde z} B_{x,z} - W_0^2\, \partial_{\tilde y}\psi\, \partial_{\tilde x} B_{x,y} + \\
& + W_0^2\, \partial_{\tilde y}\psi\, \partial_{\tilde z} B_{y,z} + W_0^2\, \partial_{\tilde z}\psi\, \partial_{\tilde x} B_{x,z} + W_0^2\, \partial_{\tilde z}\psi\, \partial_{\tilde y} B_{y,z} + \\
& + \psi(1-\psi^2) - \lambda u(1-\psi^2)^2 \\
\partial_t u = {} & \alpha\tilde\nabla^2 u + \tfrac{1}{2}\partial_{\tilde t}\psi\ , \qquad \text{on } \tilde\Omega,\ \forall \tilde t > 0\ .
\end{aligned}$$

where $\tilde{\alpha} := \frac{\alpha\tau_0}{W_0^2}$.

Time discretization The same forward Euler time discretization used for the two dimensional model is chosen:

$$\begin{aligned}
\Delta\psi^n &= \tfrac{\Delta t}{A^n}\Big\{[A^2]^n\nabla^2\psi^n + \nabla[A^2]^n\cdot\nabla\psi^n + \\
&\quad - \partial_x\psi^n\,\partial_y[B_{x,y}]^n - \partial_x\psi^n\,\partial_z[B_{x,z}]^n - \partial_y\psi^n\,\partial_x[B_{x,y}]^n + \\
&\quad + \partial_y\psi^n\,\partial_z[B_{y,z}]^n + \partial_z\psi^n\,\partial_x[B_{x,z}]^n + \partial_z\psi^n\,\partial_y[B_{y,z}]^n + \\
&\quad + \psi^n(1-[\psi^n]^2) - \tfrac{\alpha}{a_2}u^n(1-[\psi^n]^2)^2\Big\} \\
\psi^{n+1} &= \psi^n + \Delta\psi^n \\
\partial_t u^{n+1} &= u^n + \alpha\Delta t\,\nabla^2 u^n + \tfrac{1}{2}\Delta\psi^n\,, \qquad\qquad n>0\,.
\end{aligned}$$

Space discretization Let $h > 0$ be the spacing of an uniform grid $\{(x_i, y_j, z_k) : i = 0,\dots,N_x, j = 0,\dots,N_y, k = 0,\dots,N_z\}$ where $x_i := x_0 + ih$, $y_j := x_0 + jh$ and $z_k := z_0 + kh$. Naming $\psi^n_{i,j,k}$ and $u^n_{i,j,k}$ the approximations of the nondimensional temperature and phase fields at time t^n on the node (x_i, y_j, z_k), the derivatives along the z direction can be approximated by central differences:

$$\partial_z\psi^n(x_i,y_j,z_k) \approx \tfrac{\psi^n_{i,j,k+1}-\psi^n_{i,j,k-1}}{2h} \qquad \partial_z^2\psi^n(x_i,y_j,z_k) \approx \tfrac{\psi^n_{i,j,k+1}+\psi^n_{i,j,k-1}-2\psi^n_{i,j,k}}{h^2}$$

The numerical scheme for inner nodes then becomes (for $|\nabla\psi| \neq 0$):

$$\begin{aligned}
[\delta_x\psi]^n_{i,j,k} &= \tfrac{\psi^n_{i+1,j,k}-\psi^n_{i-1,j,k}}{2h} \\
[\delta_y\psi]^n_{i,j,k} &= \tfrac{\psi^n_{i,j+1,k}-\psi^n_{i,j-1,k}}{2h} \\
[\delta_z\psi]^n_{i,j,k} &= \tfrac{\psi^n_{i,j+1,k}-\psi^n_{i,j-1,k}}{2h} \\
[n^2]^n_{i,j,k} &= ([\delta_x\psi]^n_{i,j,k})^2 + ([\delta_y\psi]^n_{i,j,k})^2 + ([\delta_z\psi]^n_{i,j,k})^2 \\
[n_x]^n_{i,j,k} &= \tfrac{[\delta_x\psi]^n_{i,j,k}}{[n^2]^n_{i,j,k}} \\
[n_y]^n_{i,j,k} &= \tfrac{[\delta_y\psi]^n_{i,j,k}}{[n^2]^n_{i,j,k}} \\
[n_z]^n_{i,j,k} &= \tfrac{[\delta_z\psi]^n_{i,j,k}}{[n^2]^n_{i,j,k}} \\
a^n_{i,j,k} &= A(\varphi^n_{i,j,k}) \\
[\delta_{n_x}a]^n_{i,j,k} &= \partial_{n_x}A([n_x]^n_{i,j,k},[n_y]^n_{i,j,k},[n_z]^n_{i,j,k}) \\
[\delta_{n_y}a]^n_{i,j,k} &= \partial_{n_y}A([n_x]^n_{i,j,k},[n_y]^n_{i,j,k},[n_z]^n_{i,j,k}) \\
[\delta_{n_z}a]^n_{i,j,k} &= \partial_{n_z}A([n_x]^n_{i,j,k},[n_y]^n_{i,j,k},[n_z]^n_{i,j,k}) \\
[b_{x,y}]^n_{i,j,k} &= \tfrac{a^n_{i,j,k}}{([n^2]^n_{i,j,k})^{1/2}}([\delta_x\psi]^n_{i,j,k}[\delta_{n_y}a]^n_{i,j,k} - [\delta_y\psi]^n_{i,j,k}[\delta_{n_x}a]^n_{i,j,k})
\end{aligned}$$

$$
\begin{aligned}
[b_{x,z}]^n_{i,j,k} &= \tfrac{a^n_{i,j,k}}{([n^2]^n_{i,j,k})^{1/2}}([\delta_x\psi]^n_{i,j,k}[\delta_{n_y}a]^n_{i,j,k} - [\delta_z\psi]^n_{i,j,k}[\delta_{n_x}a]^n_{i,j,k}) \\
[b_{y,z}]^n_{i,j,k} &= \tfrac{a^n_{i,j,k}}{([n^2]^n_{i,j,k})^{1/2}}([\delta_y\psi]^n_{i,j,k}[\delta_{n_y}a]^n_{i,j,k} - [\delta_z\psi]^n_{i,j,k}[\delta_{n_x}a]^n_{i,j,k}) \\
\Delta\psi^n_{i,j,k} &= \tfrac{\Delta t}{a^n_{i,j,k}}\Big\{[a^n_{i,j,k}]^2\tfrac{\psi^n_{i+1,j,k}+\psi^n_{i-1,j,k}+\psi^n_{i,j+1,k}+\psi^n_{i,j-1,k}+\psi^n_{i,j,k+1}+\psi^n_{i,j,k-1}-6\psi^n_{i,j,k}}{h^2} + \\
&\qquad + \Big[\tfrac{[a^n_{i+1,j,k}]^2-[a^n_{i-1,j,k}]^2}{2h} - \tfrac{[b_{x,y}]^n_{i,j+1,k}-[b_{x,y}]^n_{i,j-1,k}}{2h} - \tfrac{[b_{x,z}]^n_{i,j,k+1}-[b_{x,z}]^n_{i,j,k-1}}{2h}\Big][\delta_x\psi]^n_{i,j,k} + \\
&\qquad + \Big[\tfrac{[b_{x,y}]^n_{i+1,j,k}-[b_{x,y}]^n_{i-1,j,k}}{2h} + \tfrac{[a^n_{i,j+1}]^2-[a^n_{i,j-1}]^2}{2h} - \tfrac{[b_{y,z}]^n_{i,j,k+1}-[b_{x,z}]^n_{i,j,k-1}}{2h}\Big][\delta_y\psi]^n_{i,j,k} + \\
&\qquad + \Big[\tfrac{[b_{x,y}]^n_{i+1,j,k}-[b_{x,y}]^n_{i-1,j,k}}{2h} + \tfrac{[b_{y,z}]^n_{i,j+1,k}-[b_{x,z}]^n_{i,j-1,k}}{2h} + \tfrac{[a^n_{i,j,k+1}]^2-[a^n_{i,j,k-1}]^2}{2h}\Big][\delta_z\psi]^n_{i,j,k}\Big\} + \\
&\quad + \psi^n_{i,j}(1-[\psi^n_{i,j}]^2) - \tfrac{\alpha}{a_2}u^n_{i,j}(1-[\psi^n_{i,j}]^2)^2\Big\} \\
\psi^{n+1}_{i,j} &= \psi^n_{i,j} + \Delta\psi^n_{i,j} \\
u^{n+1}_{i,j} &= u^n_{i,j} + \alpha\Delta t\nabla^2 u^n_{i,j} + \tfrac{1}{2}\Delta\psi^n_{i,j} \ , \\
&\qquad 0<i<N_x,\ 0<j<N_y,\ 0<k<N_z,\ n>0\ .
\end{aligned}
$$

Parallelization

Following the same subdivision into task described in §4.2.1, some modification are made to the two dimensional implementation:

Task 1 Compute the gradient of the phase field and its norm using the previous solution.

$$
\begin{aligned}
[\delta_x\psi]^n_{i,j,k} &= \tfrac{\psi^n_{i+1,j,k}-\psi^n_{i-1,j,k}}{2h} \\
[\delta_y\psi]^n_{i,j,k} &= \tfrac{\psi^n_{i,j+1,k}-\psi^n_{i,j-1,k}}{2h} \\
[\delta_z\psi]^n_{i,j,k} &= \tfrac{\psi^n_{i,j,k+1}-\psi^n_{i,j,k-1}}{2h} \\
[n^2]^n_{i,j,k} &= ([\delta_x\psi]^n_{i,j,k})^2 + ([\delta_y\psi]^n_{i,j,k})^2 + ([\delta_z\psi]^n_{i,j,k})^2\ .
\end{aligned}
$$

Task dependency are unchanged while $\boldsymbol{\delta_z\psi^n}$ must be added to the list of computed outputs.

Task 2 Compute the terms depending on the anisotropy function and its derivatives.

$$
\begin{aligned}
\text{if } [n^2]^n_{i,j,k} = 0\,, \qquad\qquad a^n_{i,j,k} &= 1 + \varepsilon \\
[a^2]^n_{i,j,k} &= (a^n_{i,j,k})^2 \\
[b_{x,y}]^n_{i,j,k} &= 0\,; \\
\text{otherwise,} \qquad\qquad a^n_{i,j,k} &= 1 + \varepsilon\{4\tfrac{([\delta_x\psi]^n_{i,j,k})^4+([\delta_y\psi]^n_{i,j,k})^4+([\delta_z\psi]^n_{i,j,k})^4}{([n^2]^n_{i,j,k})^2} - 3\} \\
[a^2]^n_{i,j,k} &= (a^n_{i,j,k})^2 \\
[b_{x,y}]^n_{i,j,k} &= 16\varepsilon a^n_{i,j,k}\tfrac{[\delta_x\psi]^n_{i,j,k}[\delta_y\psi]^n_{i,j,k}}{[n^2]^n_{i,j,k}}\tfrac{([\delta_x\psi]^n_{i,j,k})^2-([\delta_y\psi]^n_{i,j,k})^2}{[n^2]^n_{i,j,k}} \\
[b_{x,z}]^n_{i,j,k} &= 16\varepsilon a^n_{i,j,k}\tfrac{[\delta_x\psi]^n_{i,j,k}[\delta_z\psi]^n_{i,j,k}}{[n^2]^n_{i,j,k}}\tfrac{([\delta_x\psi]^n_{i,j,k})^2-([\delta_z\psi]^n_{i,j,k})^2}{[n^2]^n_{i,j,k}} \\
[b_{y,z}]^n_{i,j,k} &= 16\varepsilon a^n_{i,j,k}\tfrac{[\delta_y\psi]^n_{i,j,k}[\delta_z\psi]^n_{i,j,k}}{[n^2]^n_{i,j,k}}\tfrac{([\delta_y\psi]^n_{i,j,k})^2-([\delta_z\psi]^n_{i,j,k})^2}{[n^2]^n_{i,j,k}}\,.
\end{aligned}
$$

$\boldsymbol{\delta_z\psi}^n$ must be added to the dependency list of this task and also $[\boldsymbol{b_{x,z}}]^n$, and $[\boldsymbol{b_{y,x}}]^n$ must to the compute outputs one. The task still does not require information exchange between patches.

Task 3 Compute the new phase and thermal fields at next time-step.

$$
\begin{aligned}
\Delta\psi^n_{i,j,k} = \tfrac{\Delta t}{a^n_{i,j,k}}\Big\{&[a^n_{i,j,k}]^2\tfrac{\psi^n_{i+1,j,k}+\psi^n_{i-1,j,k}+\psi^n_{i,j+1,k}+\psi^n_{i,j-1,k}+\psi^n_{i,j,k+1}+\psi^n_{i,j,k-1}-6\psi^n_{i,j,k}}{h^2} + \\
&+\Big(\tfrac{[a^n_{i+1,j,k}]^2-[a^n_{i-1,j,k}]^2}{2h} - \tfrac{[b_{x,y}]^n_{i,j+1,k}-[b_{x,y}]^n_{i,j-1,k}}{2h} - \tfrac{[b_{x,z}]^n_{i,j,k+1}-[b_{x,y}]^n_{i,j,k-1}}{2h}\Big)[\delta_x\psi]^n_{i,j,k} + \\
&+\Big(\tfrac{[b_{x,y}]^n_{i+1,j,k}-[b_{x,y}]^n_{i-1,j,k}}{2h} + \tfrac{[a^n_{i,j+1,k}]^2-[a^n_{i,j-1,k}]^2}{2h} - \tfrac{[b_{y,z}]^n_{i,j,k+1}-[b_{y,z}]^n_{i,j,k-1}}{2h}\Big)[\delta_y\psi]^n_{i,j,k} + \\
&+\Big(\tfrac{[b_{x,z}]^n_{i+1,j,k}-[b_{x,z}]^n_{i-1,j,k}}{2h} + \tfrac{[b_{y,z}]^n_{i,j+1,k}-[b_{y,z}]^n_{i,j-1,k}}{2h} + \tfrac{[a^n_{i,j,k+1}]^2-[a^n_{i,j,k-1}]^2}{2h}\Big)[\delta_y\psi]^n_{i,j,k} + \\
&+\psi^n_{i,j,k}(1-[\psi^n_{i,j,k}]^2) - \tfrac{\alpha}{a_2}u^n_{i,j,k}(1-[\psi^n_{i,j,k}]^2)^2\Big\} \\
\psi^{n+1}_{i,j,k} = \psi^n_{i,j,k}& + \Delta\psi^n_{i,j,k} \\
u^{n+1}_{i,j,k} = u^n_{i,j,k}& + \alpha\Delta t\nabla^2 u^n_{i,j,k} + \tfrac{1}{2}\Delta\psi^n_{i,j,k}.
\end{aligned}
$$

To the dependency list of this task must be added the additional outputs of the previous two task. A one element thick layer of ghosts are required for variables computed by *Task 2*.

4.2.6 3D Model – Results

Since the implementation of the three dimensional model stressed out some limitations (discussed in §4.3.2) in the component implementation at this stage, the validation of this step has been postponed to the next chapters, when such limitations will be overcame.

As a demonstration of the stage progress, a simulation on a computational domain with 200^3 cells with cell-centred data and no adaptivity has been performed. In Figure 4.16 a plot of the iso-surface $\psi = 0$ corresponding to the liquid/solid interface at time $t = 340$ is presented. The corresponding nondimensional temperature field is, instead, illustrated in Figure 4.17 by means of colour plots on the planes generated by the coordinate axes.

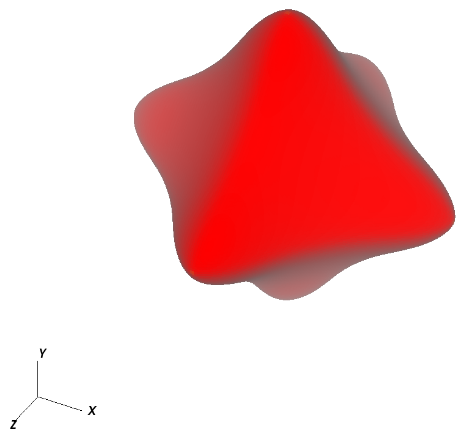


Figure 4.16: Iso-surface $\psi = 0$ at time $t = 340$. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $\Omega = [-100, 100]^3$, $h = 1$, $\Delta t = 0.1$]

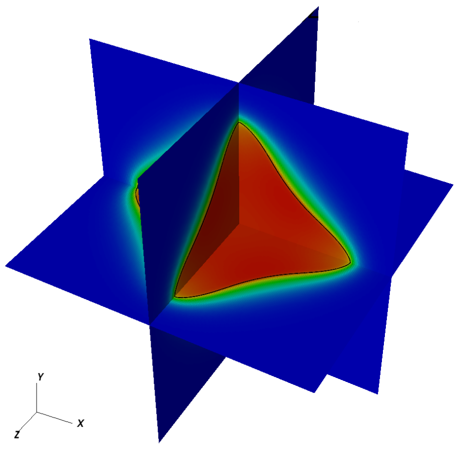


Figure 4.17: Colour plots for u on the planes generated by the coordinate axes. In black is shown the contour line for $\psi = 0$ on those planes. [$\alpha = 1$, $\varepsilon = 0.05$, $R_0 = 5$, $\Delta = 0.65$, $\Omega = [-100, 100]^3$, $h = 1$, $\Delta t = 0.1$]

4.3 Heat Uintah Component

The reason to implement a component for solving the heat diffusion problem, is to have the simplest Uintah component implementation possible. In this way it is possible to implement and more quickly new features.

4.3.1 Implementation

Initially a `Heat` component has been developed for solving problem (2.25) on uniform 2D grids with explicit time stepping. The only task implemented in the component to update the solution at each timestep is the `task_time_advance_solution`.

To distinguish between internal grid elements and elements lying on a boundary, the task, first, computes the range of internal grid elements and invokes the default version of the kernel `time_advance_solution` and then, for each boundary face, it retrieves the information about the boundary condition to apply on each face and passes it as arguments to an overridden version of the `time_advance_solution` kernel.

The default kernel use the old solution to approximate the Laplacian on each grid entry accessing directly the elements pointed by the 5 point stencil. The boundary kernel, instead, must identify which entries on the stencil are out of the computational domain (ghost elements) and select the appropriate implementation according to the boundary face and condition prescribed.

There are eight different possible cases just for grid elements on a boundary face (four faces times two boundary types). Each face case have then to distinguish between inner edge entries, where only one condition is imposed, and vertices, where two conditions may apply.

In the next step of the implementation, AMR support is added introducing the `AMRHeat` class, which inherits from the `Heat` class implementing the new component.

As described in §4.2.3 the new class overrides some of the `Application` schedulings in order to execute the refinement, coarsening and error estimation tasks. In addition to implementing these tasks and their relative kernels, the `task_time_advance_solution` has been modified to handle not only computational boundaries, but also the interfaces between coarse and refined levels. The way internal ranges are computed has been updated and also the boundary implementation of the `time_advance_solution` has been updated.

Additional implementations have been added for approximating the Laplacian at the interfaces. This approach of adding additional implementation is lacking of generality, prone to errors and very hard to maintain as discussed in the next section.

In the last step, implicit time stepping strategies are implemented in the component for uniform grids. Two additional tasks have been so implemented for the Backward–Euler and the Crank–Nicolson schemes respectively: `task_time_advance_solution_backward_euler` and `task_time_advance_solution_crank_nicholson`. For the sake of simplicity, initially only periodic domains have been supported. This feature is supported directly by the UCF, therefore only the kernels for interior grid elements have been implemented. The scheduler which scheduled the `task_time_advance_solution` has been modified to switch between the tasks depending on the user input.

As soon the implementation of boundary conditions has started, the limitations of the approach chosen for the Heat component became evident. For this reason, the changed approach with the implementation described in the next chapter, before the boundary conditions implementation was completed here, therefore this step has not been validated extensively and no results are reported here.

4.3.2 Limits

From the implementation section the lack of generality of the current approach is evident. The boundary kernel for updating the solution has many nested switches to take into account all the possible combinations of boundary conditions and patch disposition across levels. This makes the boundary kernels too complicated too maintain and extremely prone to errors. The amount of code duplication make their source very hard to maintain as well. For more complex applications, each time advance kernel will require the same ramifications of cases and switches and these must be replicated for any different phase field application that could be implemented in the future.

A second limit of the current approach is the patch partitioning into internal and boundary regions performed by time advance tasks. These partitioning is performed at every time step even if the resulting sub regions are the same since they depend exclusively on which boundary conditions the user has specified in the problem specification file, on the problem dimension, on the variable basis, and on the given stencil width.

The current approach is not generic and does not fit into the general framework of the UCF and prevents the application developer from implementing additional applications without focusing on lower level implementations.

Chapter 5

PhaseField Component General Framework

The development of a phase–field solver within Uintah until this stage resulted in two distinct components: the initial *PhaseField* and the *Heat* components. The first one implements a finite–difference solver with explicit forward Euler time stepping for the anisotropic pure metal problem (2.5) which is an actual phase-field problem; the latter, a more complete finite–difference solver for the Heat equation (2.25) which, despite not being a phase-field problem, has been used for testing new implementations.

The goal of this next development step is to merge the *Heat* component into the *PhaseField* one before their implementations diverge further. This is achieved by building a general framework which will also increase the maintainability of the code: in this way improvements made to one application will automatically apply to the others. A detailed discussion of which features are to be implemented into this framework is presented in §5.1.

In order to focus exclusively on building such a general finite–difference framework for multiple applications, of mainly phase-field models, neither boundary conditions nor implicit time stepping are to be implemented at this stage. Nevertheless, the previous implementation within the *Heat* component of these features, even if temporarily set aside, will influence the development of this general framework as described in §5.2.

No boundary condition implementation will be available within the component after this step. As a consequence, problem (2.25) is not a feasible candidate to validate the reliability of the PhaseField component. For this purpose the four phase–field models introduced in §2.1.3 have been implemented as new applications within the *PhaseField* component, and their results have been compared in [20] with those from different implementations by various researchers. In §5.3 details of this validation step are provided.

5.1 Analysis

The aforementioned developments in §4.2 and §4.3 resulted in two new and distinct components available within the UCF: *PhaseField* and *Heat*.

After the previous stage, the *PhaseField* component featured adaptive mesh refinement but no boundary–conditions enforcement: the restrictive assumption that *everything happens far from the boundaries (physical and artificial)* was made, but several features in its implementation are desirable to keep in the implementation of the general frameworks. These include:

- Grid types and functions templatization on the type of variable basis (cell–centered, `CC`, or node–centered, `NC`), which allows the application developer to implement only one class template rather than two distinct classes for using both spatial discretization techniques for the same application.

- Introduction of `View` and `ConstView` alias templates whose declarations are dependent on the data interfaces chosen at compile-time (none, Kokkos, CUDA, etc.). These take the responsibility of handling the different cases away from application developers by offering them a more abstract *adaptor*. This approach is in line with the Uintah philosophy to isolate application developers from the underlying programming model (e.g.portable loop statements: see [40]).

- Definition of finite–difference approximations as function templates with the variable base, the stencil width and the problem dimension as non-type template parameters. The application developer can use the same one class template to implement the same one application using different stencils for different problem dimensions.

- Mesh adaptivity implementation as an extension of the single grid application through class inheritance. In this way the two implementations are distinct, with two classes making it easier to distinguish between their two different work-flows and, at the same time, avoiding the need to duplicate common implementations.

The *Heat* component, on the other hand, consisted of many different applications: one for each type of variable basis and dimension. This was beneficial during the implementation of boundary condition enforcement and implicit time stepping, but resulted in multiple copies of the same code. Such an approach is not feasible as it affects the maintainability of the code when new features or bug fixes are added since they would need to be replicated for each one of the applications.

5.2 Implementation

As illustrated in §2.2, the design of a new component within the Uintah Computation Framework articulates in the implementation of three different kinds of methods. These are:

Schedulings – they are responsible for creating `Tasks`, defining which variables need to be retrieved from which data-warehouse as well as the kind and thickness of the ghost layer requested for each of them. Schedulings also define which variables are to be modified or created in the new data-warehouse. They also need to add these `Tasks` to the scheduler and specify how to parallelize their execution.

Tasks – their purpose is to retrieve input data from the data-warehouses and allocate data for output in the new data-warehouse, as well as performing all computations required to perform a simulation step. If their implementation involves iterations over grid elements they should use the adaptor loop `parallel_for`, which is an interface to the implementation of a concurrent loop. In order to do so, retrieved data and allocated variables need to be wrapped by the appropriate view and functors need to be defined for the computations to be made at each grid element.

Kernels – they implement the computation at the grid–element level. Functors for concurrent loops are instantiated as closures by means of lambda expressions.

The different schedulings are declared as virtual methods within the `ApplicationCommon` interface, one for each simulation step that form the work-flow of the chosen `SimulationController`. Their number and signature is predefined while the tasks they are adding to the scheduler are strongly dependent on the particular application. As such, no further generalization is needed for these methods. Moreover, these methods are invoked only when the `TaskGraph` needs to be compiled (initially and, for AMR simulations, after a regrid has occurred, unless otherwise specified); therefore their optimization will not be particularly beneficial.

Conversely tasks may benefit from a higher level of abstraction. The work-flow of a `Task` operating over a set of patches is often the same on each patch:

1. retrieve input data,
2. allocate output data,
3. select appropriate data wrapper,
4. iterate kernels over grid entries,
5. save reduced variables.

For the implementation of boundary conditions and mesh adaptivity for the *Heat* component using the forward Euler method it was necessary, first, to define two different kernels for updating the solution on the interior and at the boundary and, second, to re-implement a different task for updating the solution on refined regions. To switch between which kernel to update the inner solution (non–checking for out of range indices) and the boundary one, a task needed to: (a) partition the patch into an inner region and faces, (b) loop the inner kernel over the inner region, (c) for each face retrieve which boundary condition applies, and (d) iterate the boundary kernel over each face.

In addition the task on a refined region has to retrieve data from the coarser regions at all fine–coarse interfaces and, before iterating over a face at an artificial boundary, it also has to perform an appropriate interpolation of the data from the coarser level prior to invoking the refinement kernel. In this setup, boundary and refinement kernels are switching between different implementations for each possible boundary condition (faces) and combinations of boundary conditions (edges and vertices), which led to multiple nested checks and increasing code duplication with the implementation of three–dimensional problems and new boundary conditions.

Beside maintainability, also performance is an issue for this approach. In fact steps (a) and (c) are performed at each time step despite being required only on geometry changes. Even worse is the fact that checks and switches to select the correct implementation within boundary and refinement kernels are made at each time step and each boundary grid element, when also they could just be performed only when the geometry is updated.

Our approach to overcome these problems is to create a container for each patch for storing its partition into interior/faces/edges/vertices and to provide *views* to the data, offering a common interface to the various implementations. The first step towards this abstraction was to focus on the implementation of the interfaces and to temporarily set aside the implementation of patch sub-partitioning.

5.2.1 View Interfaces

Two different template interfaces have then implemented for the application developer:

`View` – which wraps the grid variable classes for accessing grid data from a data-warehouse;

`FDView` – which extends the `View` interface with pure virtual methods, allowing the abstract invocation of the various finite-difference approximations of differential operators.

Both view interfaces are templated on the type of field behind the grid data: `ScalarField<T>` for uni–dimensional physical fields of type `T` or `VectorField<T,N>` for `N`-dimensional vector fields. In addition to the field type, `FDView` is also templated on the type of stencil used for the finite–difference approximations.

All classes implementing the scalar view interface must provide the following:

– two versions of the `set` method for making the view range over either a patch (the first), or over a region (the second), where a region is any proper or improper rectangular subset of a patch;

– the `clone` method for allowing instantiation of a new copy of the implementation from the interface;

– a `get_support` method to retrieve the region over which the view is defined;

– an `is_defined_at` method to check if the view has access to the field value at a given location in the grid; and

– both a `const` and a non-`const` `operator[]` for accessing or modifying the value of the field at a given grid element.

It was considered appropriate to isolate the declarations of the *complex* differential operators – gradient and laplacian, from the implementation of the other *simple* finite–difference methods which they are dependent upon. An intermediate interface class, `detail::basic_fd_view<ScalarField<T>>`, has been introduced to provide the methods:

– `dx`, `dy`, `dz` to evaluate finite–difference approximations of the first order derivatives of the field at grid elements;

– `dxx`, `dyy`, `dzz` to approximate the second order derivatives of the field.

These interfaces are then inherited by the `FDView` interface so that its implementation must provide the definition of the following methods:

– `gradient` to get the approximate vector of the derivatives;

– `laplacian` to evaluate the discrete nabla operator.

Vector views are then defined to use the use scalar views for each of their component. This is achieved introducing the `detail::view_array` class template and using the curiously recurring template pattern (CRTP) [24]. A detailed descrption of this and the previous implementations together with the UML class diagram of the view interface are available in the manual that is provided with the source code on the researcher's GitHub [18].

5.2.2 Data-Warehouse View Implementations

The implementation of the `View` interface to be used on inner non–boundary grid elements is called `DWView` since only the data from the data-warehouse are required to perform all operations needed by the interface.

It has two non-type template parameters in addition to `View`: the variable type representation (`VAR`), and the problem dimension (`DIM`).

Since views are designed to be able to compute themselves what data to retrieve from the data-warehouse, they store references to the variable label and the material which are the indexes to look up the data data in the data-warehouse.

When `DWViews` are initialized, they can either automatically retrieve the data associated to their patch or this stage can be delayed abd the `set` method must be explicitly used for setting a view either on a whole patch or a subset rectangular region of that patch.

The actual grid region retrieved from the data-warehouse is computed according to the variable type `VAR`, the problem dimension and the number of ghost elements `GN`.

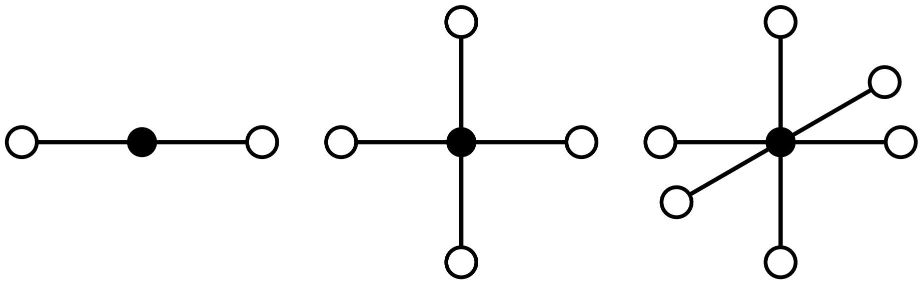

(a) P3: 3–point 1D (b) P5: 5–point 2D (c) P7: 7–point 3D

Figure 5.1: Stencil types available in the StnType enumeration.

For scalar DWViews, the get_support and is_defined_at methods are also provided to check which grid region of the domain the view has access to and if an index belongs to it. The former return an object of type Support which is generally a list of rectangular Regions, but in this case it consists of just one region whose first and past-the-end indices are obtained, calling the inner m_variable getLowIndex and getHighIndex methods.

The data-warehouse FDView implementation is named DWFDView. Its scalar specialization is derived upon the detail::dw_fd class template which provides the implementations of the methods:

```
template <DirType DIR> inline T d ( const IntVector & id ) const;
template <DirType DIR> inline T d2 ( const IntVector & id ) const;
```

for computing the approximations of the first and second order derivatives.

This is the lower level of implementation of the differential operators since these direction independent definitions are the at base of any other differential operator that may be defined for a given stencil.

At the next level in the view data-warehouse hierarchy there is the detail::dw_basic_fd_view template class with the field, stencil and variable types as parameters. It inherits the dw_fd class and defines the *direction–specific* differential operators required by the _basic_fd_view intermediate interface.

One more step up the view ladder is the detail::dw_fd_view, which implements the *complex* differential operators gradient and laplacian by calling the *simple* operations interfaces made available by the detail::basic_fd_view interface.

These implementations correspond to the three– (P3), five– (P5), and seven–point (P7) stencils respectively in one, two and three dimensions as shown in Figure 5.1 (in parenthesis are the names used to identify each stencil in the StnType enumeration of available stencils).

It is also possible to implement additional stencils by adding a new element to the StnType enumeration and implementing accordingly just the dw_fd class or also the gradient and laplacian methods of the corresponding detail::dw_fd_view class specialization.

In the PhaseField component implementation, each view implements the clone method for creating new view instances of the same class implementation from the abstract view interface without explicitly knowing its type. This will be particularly beneficial when more implementations of the view interfaces will be available for handling boundary conditions and multi–grid interfaces. For example, an AMR application may need to have access to the same refined region both, from that refined level, for updating the solution and, from coarser one, for enforcing the continuity of the solution between that level and the refined one; in this case it may be useful to have an additional view for accessing the data over that same refined region from the other level in addition to the one already instatiated to be used on the same refinement level. Being able to use the existing instance as a prototype, in this case, makes it possible to skip the need to implement again all the logic needed to choose the appropriate view implementation, which could be an expensive task if all possible topological and geometrical configurations of the specific region have to be taken into account. The `clone` methods take a flag, `deep`, as an input parameter to control if the inner variable has to be reinstantiate as well or if is enough to simply point to the one owned by the prototype.

In addition to the standard constructors and to the prototype creational pattern, it may be desirable to implement the *Factory Method Pattern* as well. This is a creational pattern designed for dealing with the problem of creating object instances without having to specify explicitly the exact class type. This paradigm is particularly useful when the choice of which implementation to instantiate is determined at run time, for example, from parsing an input file: in this case, in fact, a string can be constructed from the specifications given in the input file to uniquely identify a specific implementation.

An indirect positive consequence of this paradigm is beneficial when the factory implementation classes are templates, since their factory name has to be explicitly defined once and only once for each combination of template parameters that are used by any application, even if the factory pattern is not used for creating them. However, the definition of the factory name cases the explicit instantiation of any instance of the class template specialization it refers to.

This ensures that only one instance for each class template specialization is compiled, reducing the overall compilation time and the final size of the compiled binary files.

The inheritance diagram of the data-warehouse view implementations and a more detailed descption of the aforementioned implementation is available in [18].

5.2.3 Application Implementation

The template interface `Application` has been implemented to be used as the base class for all application implementations within the *PhaseField* component. It is templated on the variable representation type used for the model variables (`VAR`), the problem dimension (`DIM`) and the finite–difference stencil (`STN`) to be used for the spacial discretization. It publicly inherits the default ApplicationCommon interface used as base class by all other Uintah components as well as the `DWInterface<VAR,DIM>` utility class which provides static methods for browsing the grid and accessing the data-warehouse in a unified fashion (see [18] for an UML class diagram and a more in depth description of its implementation). To retrieve the coordinates relative to a given grid index, two get_position methods are provided: the first version uses a Patch object to identify which refinement level the grid index corresponds to, the second one uses directly a Level object for that purpose. The inverse of the get_position method is the find_point method. This checks if a given set of coordinates belongs to a patch and, if true, saves the corresponding index to the IntVector reference specified as input parameter and returns true, otherwise it returns false. The methods get_low, get_high and get_range are then provided to retrieve the first, past-the-end and range of indices of the grid elements belonging to a given patch. All these methods are redirecting to template interfaces within the detail namespace: detail::dw_interface0<VAR> and detail::dw_interface1<VAR,DIM>. Their template class specializations provide general named wrapper methods to implementation–specific methods within original Uintah classes; as an example the get_low definition is given below:

```
static inline IntVector
get_low ( const Patch * p )
{
  return p->getCellLowIndex();
}
```

The original *PhaseField* component implementing the anisotropic pure metal problem now becomes the PureMetal application and the various implementations of the heat component merge together into the one Heat application. At this stage, for the sake of simplicity, these applications temporarily drop the support to boundary conditions and the implicit solver. Such features will then be reintroduced and improved later in the development.

Thanks to the new interfaces, it is now possible to implement more easily the solvers for the benchmark problems (§2.1.3) as new applications within the *PhaseField* component. It is the solutions to these problems that have been used to assess the reliability of the developments introduced at this stage since they do not require the implementation of any boundary conditions (since periodic geometries are handled natively within Uintah).

An example of the improvements for the application developer is the implementation of the task_forward_euler_time_advance. This is the code used to loop over each patch in the given PatchSubset, to retrieve the old variable from the old data-warehouse, to allocate the same variable in the new data-warehouse for the updated solution, and to actually compute that solution depended on whether cell–centered or vertex–based variables were used, on the number of ghost elements required by the chosen stencil, and on the problem dimension. This is all evident in the snippet below, taken from the cell–centered, 2D implementation of the old heat component.

```
constCCVariable<double> u_old;
dw_old->get(u_old, u_label, 0, patch, Ghost::AroundCells, 1);

CCVariable<double> u_new;
```

```
dw_new->allocateAndPut(u_new, u_label, 0, patch);

BlockRange range {
  patch->getCellLowIndex() + IntVector (
     patch->getBCType ( Patch::xminus ) == Patch::Neighbor ? 0 : 1,
     patch->getBCType ( Patch::yminus ) == Patch::Neighbor ? 0 : 1,
     0 ),
  patch->getCellHighIndex() - IntVector (
    patch->getBCType ( Patch::xplus ) == Patch::Neighbor ? 0 : 1,
    patch->getBCType ( Patch::yplus ) == Patch::Neighbor ? 0 : 1,
    0 )
  };

parallel_for ( range, [patch, &u_old, &u_new, this] ( int i, int j, int k )->void {
     forward_euler_time_advance ( i, j, k, patch, get_view ( u_old ), get_view ( u_new ) ); } );
```

The same task is now simplified as below:

```
DWFDView<ScalarField<const double>,STN,VAR> u_old_view(dw_old, u_label, material, patch);
DWView<ScalarField<double>,VAR,DIM>  u_new (dw_new, u_label, material, patch);
BlockRange range ( this->get_range ( patch ) );
parallel_for ( range, [patch, &u_old, &u_new, this] ( int i, int j, int k )->void {
     time_advance_solution_forward_euler ( {i, j, k}, u_old, u_new ); } );
```

This same code can be used for all variable types, discretization stencils, and problem dimensions. The grid variable type and the ghost layer specification don't have to be explicitly coded and the appropriate choice is made by the compiler when resolving the templates according to the parameters specified.

The *.ups* input file is an xml file whose root node is named Uintah_specification. To specify that the *PhaseField* component has to be used it is necessary to set the type attribute of the SimulationComponent first level child to the string 'phasefield' and to select a particular application the type attribute of the PhaseField child of the root node is used. Possible values are 'pure_metal', 'heat', 'benchmark01', 'benchmark02', 'benchmark03, and 'benchmark04'.

in the following an overall description of all PhaseField application implementations is provided. For a more in-depth description and the UML class diagram for phase field applications see [18].

5.2.4 PureMetal Implementation

Almost the same tasks used in §4.2 for the initial *PhaseField* component have been implemented here for the PureMetal application. The previous task_initialize method has been now split into two different tasks: one, task_initialize_solution, initializes the solution variables psi and u to the initial conditions (2.10) and the anisotropy functions a, a2 and b to zero; the other, task_ initialize_grad_psi, computes the grad_psi variable from the just initialized phase field one. The task_time_advance_current_solution task has been renamed to task_time_advance_solution. The other tasks that were already implemented for the initial *PhaseField* component (task_compute_stable_timestep, task_time_advance_grad_psi, and task_time_advance_anisotropy_terms) are performing the same work as before.

For each of the tasks a scheduling method (*scheduler*) has been defined to specify which variables it requires from the previous and current timesteps

The main difference between this implementation and the previous is that the task for computing the gradient of the phase field is now performed at the end of each time step and not at the beginning of them. This will allow, when AMR is introduced, for use of the value of grad_psi from the previous timestep to estimate the error and decide which regions to refine or coarsen, avoiding the need to compute the phase field derivatives twice.

As a consequence, it is now necessary to initialize the grad_psi variable; it is for this reason that the task_initialize_grad_psi has been introduced. However it uses the same kernel as the task_time_advance_grad_psi, since the only difference between them is from which data-warehouse the phase field variable is retrieved.

Since support to boundary conditions has been dropped, just one version of the kernel for internal grid entries has been kept from the previous *PhaseField* component. Finite difference implementations are now available as methods of the FDView interface and they are not defined anymore as methods of the component itself; the application developer no longer needs to modify the component/application implementation to change the finite difference stencil being used.

5.2.5 Heat Implementation

During the merge of the multiple implementations of the old *Heat* component into the *PhaseField* component as a single application, some modifications were introduced in order to be able to use this application, not only as a testing ground for implementing new features, but also as a tool to assess their reliability quantitatively.

Instead of the hyperbolic tangent phase–field like initial condition, the following has been implemented in the initialize_solution kernel:

$$u(\boldsymbol{x}, 0) = \prod_{i=0}^{d-1} \cos \frac{\pi x_i}{2L}, \qquad \forall \boldsymbol{x} \in \Omega \subset \mathrm{R}^d \,, \tag{5.1}$$

where $L \in \mathbb{R}^+$ is a positive constant that represent the minimal distance from each coordinate axes at which the initial condition is null and $d = 1, 2, 3$ is the problem dimension.

When the domain is $\Omega = [-L, L]^d$ and homogeneous Dirichlet boundary conditions are chosen the solution is known explicitly:

$$u(\boldsymbol{x},t) = \exp(-da^2t)\prod_{i=0}^{d-1}\cos(ax_i), \qquad \forall \boldsymbol{x}\in\Omega, t>0\,, \tag{5.2}$$

where $a := \frac{\pi}{2L}$ is the angular frequency of each cosinusoidal component of the solution. The same solution is obtained if the domain is restricted to the first octant ($\Omega = [0,L]^d$) and homogeneous Neumann conditions are imposed at $x_i = 0$ while keeping homogeneous Dirichlet conditions at $x_i = L$ ($i = 0,\dots,d-1$).

The task `task_time_advance_solution_error` has then been introduced to provide measures of the difference, ε^N, between the analytical and the computed solution at a given timestep $t^N = kN$ with $N = 1,2,\dots$, where $k > 0$ is the temporal discretization step.

The square of the L^2-norm of the solution u at a fixed time is computed by observing that variables in (5.2) are separated:

$$\begin{aligned}
\|u(\cdot,t)\|_0^2 := \|u(\cdot,t)\|_{L^2(\Omega)}^2 &:= \int_\Omega [u(\boldsymbol{x},t)]^2\,\mathrm{d}\boldsymbol{x} = \exp(-2da^2t)\int_{[-L,L]^d}\prod_{i=0}^{d-1}\cos^2(ax_i)\,\mathrm{d}\boldsymbol{x}\\
&= \exp(-2da^2t)\prod_{i=0}^{d-1}\int_{-L}^{L}\cos^2(a\xi)\,\mathrm{d}\xi\\
&= \exp(-2da^2t)\left[L+\frac{\sin(2aL)}{2a}\right]^d, \qquad t\geq 0\,,
\end{aligned}$$

which gives the following expression:

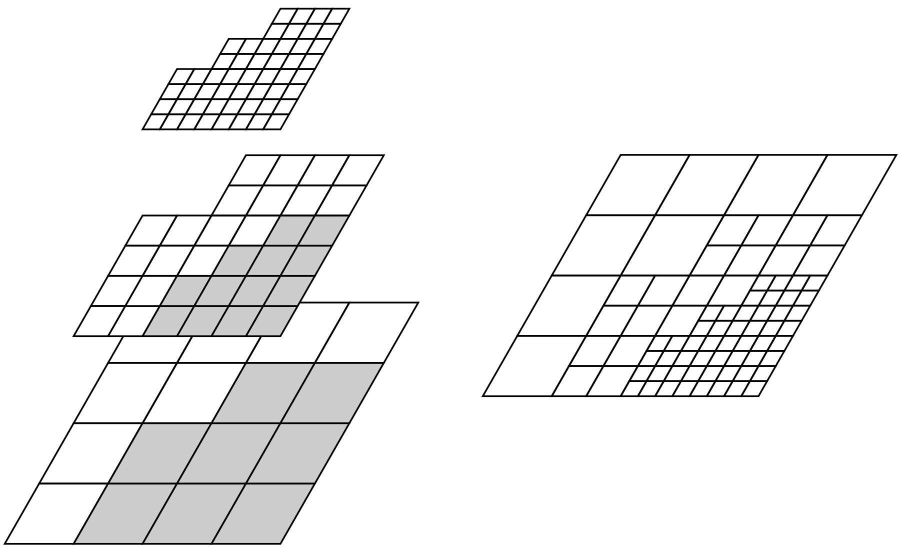
$$\|u(\cdot,t)\|_0^2 = \left[L\exp(-2a^2t)\right]^d, \qquad t\geq 0\,. \tag{5.3}$$

For each timestep $t^N = kN$, let the *spatial discretization*, $\mathscr{T}^N$, at t^N be defined as the set of grid cells which have no coarser cells below them. $\mathscr{T}^N$ is a partition of the computational domain Ω (see Figure 5.2). Each grid cell is rectangular and can be expressed as the Cartesian product of real compacts $I_{ji}^N = [x_{ji0}^N, x_{ji1}^N]$, for $i = 0,\dots,d-1$. Chosen an ordering over $\mathscr{T}^N$, all cells can be denoted as Ω_j^N, with $j = 0,\dots,M_c^N - 1$ where M_N is the cardinality of the spatial discretization.

With $\mathscr{T}_{\max}$, is denoted instead the uniform grid corresponding to the maximum level of refinement, which does not depend on the given timestep. It can be observed that any fine cell $\omega\in\mathscr{T}_{\max}$ is a subset of only a given grid cell $\Omega^N\in\mathscr{T}^N$ and that any grid cell can be expressed as the union of fine cells.

When using a *cell-centered* approximation, we can denote with $\boldsymbol{u}^N := (u_j^N)_{j=0}^{M_c^N-1}\in\mathbb{R}^{M_c^N}$ the vector of the computed solution at the N-th timestep over $\mathscr{T}^N$.

The following expression is used to define a *local L^2-discrete norm* of a discrete vector



(a) Grid with 3 refinement levels. **(b)** Corresponding partition $\mathscr{T}^N$.

Figure 5.2: Example of how to compute the partition $\mathscr{T}^N$ (b) from an adaptive grid (a). In grey the cells which have been refined and lay below a finer cells; $\mathscr{T}^N$ is then the set of all cells which have not been greyed.

$\boldsymbol{v} \in \mathbb{R}^{M_c^N}$ over the cell Ω_j^N:

$$\|\boldsymbol{v}\|^2_{0c,\Omega_j^N} := \sum_{\substack{\omega\in\mathscr{T}_{\max}\\ \omega\subset\Omega_j^N}} |\omega|(\pi_\omega \boldsymbol{v})^2, \qquad j = 0,\dots,M_c^N-1,\ N = 0,1,\dots\,.$$

where π_ω is a given interpolation operator to approximate over ω the discrete field over $\mathscr{T}^N$ represented by $\boldsymbol{v}$.

A global L^2-discrete norm can is then defined as the sum over $\mathscr{T}^N$ of the local discrete norms:

$$\|\boldsymbol{v}\|^2_{0c,\mathscr{T}^N} := \sum_{j=0}^{M_c^N-1} \|u\|^2_{0c,\Omega_j^N}, \qquad N = 0,1,\dots\,.$$

For simulations with just one level of refinement, $\mathscr{T}^N$ coincides with $\mathscr{T}_{\max}$ and Ω_j^N is the only fine cell contained in itself, therefore the operator $\pi_{\Omega_j^N}$ is chosen to return the j-th component of $\boldsymbol{v}$ and the previous expressions simply to

$$(5.4) \qquad \|\boldsymbol{v}\|^2_{0c,\Omega_j^N} := |\Omega_j^N| v_j^2 \qquad \|\boldsymbol{v}\|^2_{0c,\mathscr{T}^N} := \sum_{j=0}^{M_c^N-1} |\Omega_j^N| v_j^2, \qquad N = 0,1,\dots\,.$$

When using a *vertex-based* approximation, there may be more vertices than cells. Let $\mathscr{V}_j^N$

be the set of vertices of the grid cell Ω_j^N ($j = 0, \dots, M_c^N - 1$). We define $\mathscr{V}^N = \bigcup_{j=0}^{M_c^N-1} \mathscr{V}_j^N$ to be the set of all vertices in the grid and M_v^N to be its cardinality. With this representation, the computed solution at the N-th timestep over $\mathscr{T}^N$ can be represented as a vector $\boldsymbol{u}^N := (u_j^N)_{j=0}^{M_v^N-1} \in \mathbb{R}^{M_v^N}$.

With $\mathscr{V}_{\max}$, is now denoted the set of all vertices of the uniform grid corresponding to the maximum level of refinement. The expression used to define a *local L^2-discrete norm* of a discrete vector $\boldsymbol{v} \in \mathbb{R}^{M_c^N}$ over the cell Ω_j^N is defined as:

$$\|\boldsymbol{v}\|_{0v,\Omega_j^N}^2 := \sum_{\substack{\boldsymbol{x} \in \mathscr{V}_{\max} \\ \boldsymbol{x} \in \Omega_j^N}} \frac{A}{w_j^N(\boldsymbol{x})} (\pi_{\boldsymbol{x}} \boldsymbol{v})^2, \qquad j = 0, \dots, M_c^N - 1,\ N = 0, 1, \dots .$$

where $\pi_{\boldsymbol{x}}$ is a given interpolation operator to approximate at $\boldsymbol{x}$ the discrete field over $\mathscr{T}^N$ represented by $\boldsymbol{v}$, A is the measure of any finest cell, and $w_j^N(\boldsymbol{x})$ is a weight that takes into account the position of $\boldsymbol{x}$ in $\Omega_j^N = \prod_{i=0}^{d-1}$. If we denote with

$$\operatorname{codim}_{\Omega_j}(\boldsymbol{x}) := \left|\left\{(i,s) : x_i = x_{jis}^N, i = 0, \dots, d-1, s = 0, 1\right\}\right|$$

the number of faces of the cell Ω_j^N which the vertex $\boldsymbol{x} = (x_i)_{i=0}^{d-1}$ belong to, the weight $w_j^N(\boldsymbol{x})$ is defined to be

$$w_j^N(\boldsymbol{x}) := 2^{\operatorname{codim}_{\Omega_j^N}(\boldsymbol{x})} .$$

A global L^2-discrete norm can is then defined as the sum over $\mathscr{T}^N$ of the local discrete norms:

$$\|\boldsymbol{v}\|_{0v,\mathscr{T}^N}^2 := \sum_{j=0}^{M_c^N-1} \|u\|_{0v,\Omega_j^N}^2, \qquad N = 0, 1, \dots .$$

For non boundary vertex, $\boldsymbol{x} \notin \partial\Omega$, $w_j^N(\boldsymbol{x})$ is the number of grid cells sharing the vertex $\boldsymbol{x}$; therefore, for simulations with just one level of refinement, it is possible to rewrite the previous expressions for the global L^2-discrete norm as

$$\|\boldsymbol{v}\|_{0v,\mathscr{T}^N}^2 := \sum_{j=0}^{M_v^N-1} \frac{A}{w_j} v_j^2, \qquad N = 0, 1, \dots , \tag{5.5}$$

having observed that $\mathscr{V}_j^N \equiv \mathscr{V}_{\max}$ and that, for each $\boldsymbol{x}_j \in \mathscr{V}_{\max}$, the interpolator $\pi_{\boldsymbol{x}_j}$ can chosen to be defined by $\pi_{\boldsymbol{x}_j}(\boldsymbol{v}) = v_j$. The weight $w_j = \sum_{\substack{\omega \in \mathscr{T}_{\max} \\ \boldsymbol{x}_j \in \omega}} w_j^N(\boldsymbol{x}_j)$ can be interpreted as a factor taking into account the position of the vertex $\boldsymbol{x}_j$ with respect to the computational domain. For each vertex holds $w_j = 2^{\operatorname{codim}(\boldsymbol{x}_j)}$, where $\operatorname{codim}(\boldsymbol{x}_j)$ is the number of non-periodic boundary faces which the vertex belongs to, which coincides with the number of boundary conditions imposed on that vertex.

A measure of the global approximation error at timestep t^N is then the L^2-discrete norm of the residual vector, $\boldsymbol{\varepsilon}^N$, at t^N,

$$\varepsilon_j^N := u_j^N - u(\boldsymbol{x}_j, t^N), \qquad j = 0, 1, \dots, M^N,\ N = 0, 1, \dots\ ,$$

where $\boldsymbol{u}^N$ is the vector representing of the computed solution (either cell-centered or vector-based) at the timestep t^N. Hereafter, for uniformity of notations and where not ambiguous, the subscripts c and v will be omitted.

This measure of the approximation error, however, is not suitable for comparing results from different AMR strategies or maximum refinement levels. Moreover, since the grid $\mathscr{T}^N$ changes over time and so the error definition itself, this choice will make not trivial understanding the evolution of the error with time. This issue can be resolved by choosing a uniform reference grid $\widetilde{\mathscr{T}}$ whose set of vertices $\widetilde{\mathscr{V}}$ is a superset of $\mathscr{V}_{\max}$ for each simulation we want to be able to compare and using the L^2-discrete norm over $\widetilde{\mathscr{T}}$ as a measure of the global approximation error. In order to use this norm, it is necessary to evaluate the difference between the computed and the analytical solutions over the reference grid elements and an interpolation operator, $\widetilde{\pi}$, has to be chosen so that it is possible to define the residual $\widetilde{\boldsymbol{\varepsilon}}^N \in \mathbb{R}^{\widetilde{M}}$

$$\widetilde{\varepsilon}_{\tilde{\jmath}}^N := \widetilde{\pi}\boldsymbol{u}^N(\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}}) - u(\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}}, t^N), \qquad \tilde{\jmath} = 0, 1, \dots, \widetilde{M},\ N = 0, 1, \dots\ , \tag{5.6}$$

where an ordering over $\widetilde{\mathscr{G}}$ ($\widetilde{\mathscr{G}} = \widetilde{\mathscr{T}}$ or $\widetilde{\mathscr{G}} = \widetilde{\mathscr{V}}$, depending on the variable basis) is given, $\widetilde{M}$ is its cardinality, and $\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}}$ is the coordinate of the $\tilde{\jmath}$-th grid element, $\widetilde{\eta}_{\tilde{\jmath}}$. For cell-centered variables $\widetilde{\eta}_{\boldsymbol{j}} = \widetilde{\omega}_j$ and $\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}}$ is its centroid; otherwise, $\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}} = \widetilde{\boldsymbol{x}}_{\tilde{\jmath}}$ and $\widetilde{\eta}_{\tilde{\jmath}} = \{\widetilde{\boldsymbol{x}}_{\tilde{\jmath}}\}$.

The *(global) L^2-error* (referred to the spatial discretization $\widetilde{\mathscr{T}}$) at timestep t^N is so defined to be

$$\mathrm{err}_{0,\widetilde{\mathscr{T}}}^N := \|\widetilde{\boldsymbol{\varepsilon}}^N\|_{0,\widetilde{\mathscr{T}}}, \qquad N = 0, 1, \dots\ , \tag{5.7}$$

which, since $\mathscr{T}^N$ is uniform, can be written as

$$\left(\mathrm{err}_{0,\widetilde{\mathscr{T}}}^N\right)^2 = \sum_{\tilde{\jmath}=0}^{\widetilde{M}-1} \alpha_{\tilde{\jmath}}\, \widetilde{\varepsilon}_{\tilde{\jmath}}^2, \qquad N = 0, 1, \dots\ ,$$

which is a general form of both (5.4) and (5.5), where setting $\alpha_{\tilde{\jmath}} \equiv \widetilde{A}$ (with $\tilde{A}$ the measure of a cell in $\widetilde{\mathscr{T}}$), gives the first and $\alpha_{\tilde{\jmath}} := \widetilde{A}/w_{\tilde{\jmath}}$ the latter equation.

By grouping the terms in the previous summation conveniently it is possible to express the global L^2-error as the sum of local terms over each cell in $\mathscr{T}^N$

$$(5.8)\qquad \left(\mathrm{err}^N_{0,\widetilde{\mathscr{T}}}\right)^2 = \sum_{\Omega^N_j \in \mathscr{T}^N} \left[\sum_{\substack{\tilde{\eta}_{\tilde{j}} \in \widetilde{\mathscr{T}} \\ \tilde{\eta}_{\tilde{j}} \subset \Omega^N_j}} \alpha_{\tilde{j}} \tilde{\varepsilon}^2_{\tilde{j}} \right] = \sum_{j=0}^{M^N_c - 1} |\Omega^N_j| \left(\mathrm{err}^N_{0,j}\right)^2, \qquad N = 0, 1, \dots ,$$

with $\mathrm{err}^N_{0,j} := |\Omega^N_j|^{-1} \left[\sum_{\tilde{j}} \alpha_{\tilde{j}} \tilde{\varepsilon}^2_{\tilde{j}} \right]^{\frac{1}{2}}$.
The vector of these terms, $\mathbf{err}^N_0 := (\mathrm{err}^N_{0,j})_{j=0}^{M^N_c - 1}$, will be referred to as the vector of the local L^2-errors at t^N, which is a cell-centered vector independently on the variable basis of $\boldsymbol{u}^N$.

The `task_time_advance_solution_error` is saving $\boldsymbol{\varepsilon}^N$ to the variable epsilon_u and the local error, $\mathbf{err}^N_0$, to error_u. It is also using the reduction variables error_norm2_L2 and u_norm2_L2 to store the square of the global L^2-error at each timestep and the square of the global L^2-discrete norm for being compared with (5.3).

When the preprocessor variable `PhaseField_Heat_DBG_DERIVATIVES` is defined, also the tasks `task_time_advance_dbg_derivatives` and `task_time_advance_dbg_derivatives` are compiled and scheduled to be executed during a simulation.

The H^1_0- and H^2_0-norms[1] of the solution at $t > 0$ are computed, analogously to the L^2 one:

$$\begin{aligned}(5.9)\quad \|u(\cdot,t)\|_1^2 &:= \|u(\cdot,t)\|^2_{H^1_0(\Omega)} := \|\nabla u(\cdot,t)\|^2_{[L^2(\Omega)]^d}\,\mathrm{d}\boldsymbol{x} \\ &= \int_\Omega |\nabla u(\boldsymbol{x},t)|^2 = \int_\Omega \sum_{i=0}^{d-1} |\partial_i u(\boldsymbol{x},t)|^2\,\mathrm{d}\boldsymbol{x} \\ &= a^2 \exp(-2da^2 t) \sum_{i=0}^{d-1} \left\{ \left(\int_{-L}^{L} \sin^2(ax_i)\,\mathrm{d}x_i \right) \left(\prod_{\substack{j=0 \\ j\neq i}}^{d-1} \int_{-L}^{L} \cos^2(ax_j)\,\mathrm{d}x_j \right) \right\} \\ &= a^2 \exp(-2da^2 t) \sum_{i=0}^{d-1} \left\{ L \prod_{\substack{j=0 \\ j\neq i}}^{d-1} L \right\} = da^2 [L \exp(-2a^2 t)]^d \\ &= da^2 \|u(\cdot,t)\|_0^2\end{aligned}$$

$$\begin{aligned}(5.10)\quad \|u(\cdot,t)\|_2^2 &:= \|u(\cdot,t)\|^2_{H^2_0(\Omega)} := \|\Delta u(\cdot,t)\|_0^2 \\ &= \int_\Omega |\Delta u(\boldsymbol{x},t)|^2\,\mathrm{d}\boldsymbol{x} = \int_\Omega \left| \sum_i \partial_i^2 u(\boldsymbol{x},t) \right|^2 \mathrm{d}\boldsymbol{x} \\ &= d^2 a^4 \int_\Omega |u(\boldsymbol{x},t)|^2\,\mathrm{d}\boldsymbol{x} = d^2 a^4 \|u(\cdot,t)\|_0^2, \qquad t \geq 0\,,\end{aligned}$$

[1] Note that it is legitimate to embed the solution at each time, $u(\cdot,t)$, in the functional subspaces $H^1_0(\Omega)$ and $H^2_0(\Omega)$ only when Dirichlet conditions are imposed on at least a portion the domain boundary $\partial\Omega$, otherwise only the general Sobolev spaces $H^1(\Omega)$ and $H^2(\Omega)$ can be used. In the latter case the Poincaré inequality does not hold and $\|\cdot\|_1$ and $\|\cdot\|_2$ as defined here are seminorms rather than norm.

Let $\mathrm{D}^1_i u^N_j$ and $\mathrm{D}^2_{ii'} u^N_j$ denote a finite–difference approximation of the first and second order spatial derivatives along the i-th (and i'-th) coordinate direction(s) of the numerical solution $\boldsymbol{u}^N$ referred to the j-th grid element at the N-th timestep.
The notations $\mathbf{D}^1\boldsymbol{u}^N := (\mathrm{D}^1\boldsymbol{u}^N_j)_{j=0}^{M^N-1}$ and $\mathbf{D}^2\, u^N := (\mathrm{D}^2 u^N_j)_{j=0}^{M^N-1}$, where

$$\mathrm{D}^1 u^N_j := \left|(\mathrm{D}^1_i u^N_j)_{i=0}^{d-1}\right| = \left[\sum_{i=0}^{d-1}\left(\mathrm{D}^1_i u^N_j\right)^2\right]^{\frac{1}{2}}$$

$$\mathrm{D}^2 u^N_j := \sum_{i=0}^{d-1} \mathrm{D}^2_{ii} u^N_j,$$

$$j = 0,\dots,M^N,\ N = 0,1,\dots,$$

are introduced for the M^N–dimensional *euclidean norm of the discrete-gradient* and *discrete-laplacian* of $\boldsymbol{v}$ over $\mathscr{G}^N$ ($\mathscr{G}^N = \mathscr{T}^N$ or $\mathscr{G}^N = \mathscr{V}^N$, depending on the variable basis of $\boldsymbol{v}$).
With $\mathrm{D}^1_i\boldsymbol{u}^N := (\mathrm{D}^1_i u^N_j)_{j=0}^{M^N-1}$ and $\mathrm{D}^2_i\boldsymbol{u}^N := (\mathrm{D}^2_{ii} u^N_j)_{j=0}^{M^N-1}$, instead, we denote the M^N–dimensional vectors representing the discrete approximations of the first and second order derivatives of the computed solution at t^N over the grid $\mathscr{T}^N$.

In analogy to the continuous case, it is possible to define the *local* and *global* H^p_0*-discrete norms* ($p = 1,2$) for a discrete vector $\boldsymbol{v} \in \mathbb{R}^{M^N}$ as:

$$\|\boldsymbol{v}\|^2_{p,\Omega^N_j} := \|\,\mathbf{D}^p\boldsymbol{v}\|^2_{0,\Omega^N_k}\,, \qquad \|\boldsymbol{v}\|^2_{p,\mathscr{T}^N} := \sum_{j=0}^{M^N-1}\|\boldsymbol{v}\|^2_{p,\Omega^N_j},$$

$$j = 0,\dots,M-1,\ N = 0,1,\dots\,.$$

Using the p–residual matrices, $\boldsymbol{\varepsilon}^{N,p} \in \mathbb{R}^{M^N\times d}$, over the computational grid $\mathscr{T}^N$ at t^N,

$$\left(\boldsymbol{\varepsilon}^{N.p}\right)_{j,i} := D^p_i u^N_j - \partial^p_i u(\boldsymbol{\xi}_j, t^N), \qquad j = 0,1,\dots,M^N,\ N = 0,1,\dots\,,$$

to evaluate the approximation error of the solution derivatives would, as for the approximation of the solution itself, make it difficult to both evaluate its evolution over time and to compare different simulations.

A measure of the error in the approximation of the first and second order derivatives of the solution is then computed from the derivatives *p–residuals*, $\widetilde{\boldsymbol{\varepsilon}}^{N,p}_i$ over the same reference grid $\widetilde{\mathscr{T}}$ introduced for (5.6):

$$\widetilde{\varepsilon}^{N,p}_{i,\tilde{\jmath}} := \widetilde{\pi}\,\mathrm{D}^p_i\boldsymbol{u}^N(\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}}) - \partial^p_i u(\widetilde{\boldsymbol{\xi}}_{\tilde{\jmath}}, t^N),$$

$$i = 0,\dots,d-1,\ \tilde{\jmath} = 0,\dots,\widetilde{M},\ N = 0,1,\dots$$

The *(local) p–errors*, $\mathbf{err}^N_p \in \mathbb{R}^{M^N_c\times d}$, (referred to the spatial discretization $\widetilde{\mathscr{T}}$ at timestep t^N are then defined, as in (5.8), as matrices for which the L^2–discrete norm (referred to the computational grid $\mathscr{T}^N$) of the i-th column is equal to the L^2–discrete norm (referred to the reference grid $\widetilde{\mathscr{T}}$) of the corresponding p–residual ($p = 1, 2$)

$$|\Omega_j^N| \big(\mathrm{err}_p^N\big)_{j,i}^2 := \sum_{\substack{\tilde\eta_{\tilde\jmath} \in \widetilde{\mathscr{T}} \\ \tilde\eta_{\tilde\jmath} \subset \Omega_j^N}} \alpha_{\tilde\jmath} \big(\widetilde{\varepsilon}_{i,\tilde\jmath}^{N,p}\big)^2,$$
$$i = 0, \dots, d-1,\ j = 0, \dots, M_c^N,\ N = 0, 1, \dots$$

In analogy to the vectors $\mathbf{D}^p\,\boldsymbol{u}^N$ introduced to define the H_0^p–discrete norms, the vectors $\widetilde{\boldsymbol{\varepsilon}}^{N,p}$ are introduced

$$\widetilde{\varepsilon}_{\tilde\jmath}^{N,1} := \big|\big(\varepsilon_{i,\tilde\jmath}^{N,1}\big)_{i=0}^{d-1}\big| = \left[\sum_{i=0}^{d-1} \big(\varepsilon_{i,\tilde\jmath}^{N,1}\big)^2\right]^{\frac{1}{2}}$$
$$\widetilde{\varepsilon}_{\tilde\jmath}^{N,2} := \sum_{i=0}^{d-1} \varepsilon_{i,\tilde\jmath}^{N,2},$$
$$\tilde\jmath = 0, \dots, \widetilde{M},\ N = 0, 1, \dots,$$

so that the *(global)* H_0^p*–errors* (referred to the spatial discretization $\widetilde{\mathscr{T}}$ at timestep t^N can be defined as

$$\mathrm{err}_{p,\widetilde{\mathscr{T}}}^N := \|\widetilde{\boldsymbol{\varepsilon}}^{N,p}\|_{0,\widetilde{\mathscr{T}}}, \qquad N = 0, 1, \dots . \tag{5.11}$$

The `task_time_advance_dbg_derivatives` uses the solution computed at the previous timestep to compute and save into the du and duu views, the finite–difference approximations of first and second order spatial derivatives $\mathrm{D}_i^p\,\boldsymbol{u}^{N-1}$.

The `task_time_advance_solution_dbg_derivatives_error` is then saving the residuals $\boldsymbol{\varepsilon}^{N,p}$ and local derivatives errors $\mathbf{err}_p^N$, in the DataWarehouse. Reduction error are then used to store the square of the global H_0^p-error at each timestep and the square of the global H_0^p-norm to be compared with (5.9, 5.10).

Since it is meaningful to evaluate the numerical against the analytical solution (5.2) only for particular geometries and boundary conditions, the scheduling of `task_time_advance_solution_error` and `task_time_advance_solution_dbg_derivatives_error` is conditioned to the value of the node test within the problem specification file.

All different implementation of tasks within the previous *Heat* component are now merged together.

5.2.6 Allen–Cahn Implementation Benchmark01

A solver for problem (2.11, 2.12, 2.13) has been implemented as the `Benchmark01` application within the *PhaseField* component. For its discretization, spatial derivatives have been approximated with finite–differences ($\boldsymbol{\Delta}$ denotes the operator approximating the laplacian) which leads to the system

$$\partial_t \boldsymbol{u}^N = \varepsilon^2 \boldsymbol{\Delta} \boldsymbol{u}^N - (\boldsymbol{u}^N + \mathbf{1}) \circ (\boldsymbol{u}^N - \mathbf{1}) \circ \boldsymbol{u}^N, \qquad \text{on } \mathscr{T}^N,\ N = 1, 2, \ldots$$

where $\mathbf{1} := (1, 1, \ldots, 1)^t \in \mathbb{R}^M$, and $\cdot\circ\cdot$ denotes the Hadamard product. When the Explicit Euler scheme is used to approximate the temporal derivative we obtain the fully discrete problem:

$$\boldsymbol{u}^{N+1} = \boldsymbol{u}^N + k \left[\varepsilon^2 \boldsymbol{\Delta} \boldsymbol{u}^N - (\boldsymbol{u}^N + \mathbf{1}) \circ (\boldsymbol{u}^N - \mathbf{1}) \circ \boldsymbol{u}^N \right], \qquad \text{on } \mathscr{T}^N,\ N = 1, 2, \ldots \tag{5.12}$$

Since the benchmark value is computed by interpolation of the solution of (5.12) at $\boldsymbol{c} = (\pi, \pi)^t$, two tasks have been implemented for time advancing the simulation:

`task_time_advance_solution` which, for each given patch: (1) instantiates a `DWFDView` to the u variable in the old data-warehouse retrieving the data required; (2) instantiates a `DWView` allocating the memory required for storing the updated value of solution; and (3) iterates over the grid entries in that patch using the `parallel_for` *adaptor* with the `task_time_advance_solution` kernel.

`task_time_advance_postprocess` which computes two reduction variables using the solution computed by the previous task: `u0` and `energy`, which are the solution value at the center of the domain, $\boldsymbol{c}$, and the free energy of the system (2.14), the first of which is used for benchmark computations.

Two more tasks are defined, one for computing the initial solution and the other to advance the timestep. As usual, their list is given below together with their scheduling, kernel methods, and dependencies.

The `task_time_advance_solution` is not much different from its analogue in the *Heat* application: it loops on all given `patches` as specified by the scheduler and retrieves for each one of them the data from the old data-warehouse, `dw_old`, through an instance of `DWFDView` over a constant `ScalarField`. It then allocates memory for the updated solution in the new data-warehouse, `dw_new`, using an instance of `DWView` of a non constant `ScalarField`. Finally, it iterates over the grid elements within each patch using the `parallel_for` adaptor with the `time_advance_solution` kernel functor.

On the other hand, in the `task_time_advance_post-process` definition, Uintah's reduction variables are used for the first time. During the problem setup, the variable type to be used for the solution at $(\pi, \pi)^t$, `u0` and the free energy of the system `energy` has been set to `sum_vartype` when defining their labels

```
u0_label = VarLabel::create ( "u0", sum_vartype::getTypeDescription() );
energy_label = VarLabel::create ( "energy", sum_vartype::getTypeDescription() );
```

which is the type used for global variables whose value is depending only on time and whose global value is computed as the sum of local patch values, one for each patch. These local values, of type double, have to be computed within a task and then added to the global value in the data-warehouse via the put method as shown in the code below. Here, the two reduction variables are used differently: u0 is used to save a punctual value that may or may not belong to a patch; energy is instead used for computing a spatial integral. The first is regarded as the sum of the solution values at a specific point if that point corresponds to a grid element in the patch, or zero otherwise; the latter is itself a sum over the patch grid elements which has to be computed using the parallel_reduce_sum instead of the parallel_for adaptor.

Since the reduction operator is designed not to distinguish between patches that have been further refined and those that have not, it is the application developer's responsibility to avoid adding the same term twice. For example if adaptivity was implemented and the region around $(\pi,\pi)^t$ was refined, the current implementation would give the value of the solution at the center as the actual solution times the number of refinement levels at that point.

The kernel implementations mirror the Allen–Cahn discretization (5.12) and approximate the energy integral (2.14).

```
template<VarType VAR, StnType STN>
void Benchmark01<VAR, STN>::time_advance_solution (
  const IntVector & id,
  const FDView < ScalarField<const double>, STN > & u_old,
  View< ScalarField<double> > & u_new
)
{
  const double & u = u_old[id];
  double lap = u_old.laplacian ( id );
  double src = u * ( u * u - 1. );
  double delta_u = delt * ( epsilon * epsilon * lap - src );
  u_new[id] = u + delta_u;
}
```

```
template<VarType VAR, StnType STN>
void Benchmark01<VAR, STN>::time_advance_post-process_energy (
  const IntVector & id,
  const Patch * patch,
  const FDView < ScalarField<const double>, STN > & u_new,
  double & energy
)
```

```
{
  const double & u = u_new[id];
  auto grad = u_new.gradient ( id );
  double A = patch->getLevel()->dCell() [0] * patch->getLevel()->dCell() [1];
  energy += A * ( epsilon * epsilon * ( grad[0] * grad[0] + grad[1] * grad[1] ) / 2.
      + ( u * u * u * u - 2 * u * u + 1. ) / 4. );
}
```

5.2.7 Cahn–Hilliard Implementations Benchmark02, Benchmark03, and Benchmark04

Three different Cahn–Hilliard problems are implemented within the *PhaseField* component: Benchmark02, a solver for the *seven circles* problem (2.15, 2.17, 2.18); Benchmark03, for the one dimension problem (2.15, 2.19, 2.20); and Benchmark04, for the energy decay problem (2.15, 2.21, 2.22). In order to compute the approximation of the biharmonic operator Δ^2, an auxiliary variable, $v = \nabla u$, is defined and the original problem is split into a system of two equations

$$\begin{cases} \partial_t u = \Delta v \\ \quad v = -\varepsilon^2 \Delta u + W'(u) \end{cases}$$

When finite–differences are introduced for the spatial derivatives and Explicit Euler time stepping is introduced, the following discrete problem is obtained

$$\begin{cases} \boldsymbol{u}^{N+1} = \boldsymbol{u}^N + k\boldsymbol{\Delta v}^N \\ \quad v^N = -\varepsilon^2 \boldsymbol{\Delta u}^N + (\boldsymbol{u}^N + \mathbf{1}) \circ (\boldsymbol{u}^N + \mathbf{1}) \circ \boldsymbol{u}^N \end{cases}$$

The task for advancing the solution is split as well in two distinct tasks: task_time_advance_v and task_time_advance_u. The first is computing the current value of the v variable as the discrete laplacian of the solution at the previous timestep, while the second is using both the newly computed value of v and the old value of u to compute the new solution. Since the view implementations are general with respect to the problem dimension, these tasks, as well as their kernels, are defined identically in all applications, despite the Benchmark03 problem dimension being different from the others.

The task for advancing the intermediate variable v listed above is similar to the one advancing the problem variable u whose code can be found hereafter. Their structures are similar: there is an outer loop over the patches assigned to the current process by the scheduler, then the views for accessing the variables required by the task are instantiated, than the kernels are executed over the cells in the range of each patch using the parallel_for adaptor.

The main difference between the task implementations above is in the initialization of the views required by the kernels. Using the correct view type is fundamental in controlling if a view can modify or just read the values from the data warehouse and whether ghost nodes are required. On one hand, the kernel `time_advance_v` computes the value of `v_new` using the laplacian of `u_old`, so `u_old` field is not modified but requires ghost nodes, while `v_new` needs to write to the data warehouse but does not need ghost nodes. Therefore, the appropriate types are `DWFDView<const double>`, for `u_old`, and `DWView<double>`, for `v_new`. The kernel `time_advance_u` , on the other hand, uses the value of `u_old` and the laplacian of `v_new` to compute the value of `u_new`, so the types used for the variables in `task_time_advance_u` are `FDView<const double>`, for `u_old`, `DWFDView<const double>`, for `v_new`, and `DWView<double>`, for `u_new`.

The differences in the implementation of these benchmark applications are only in the initial conditions and in the quantities used for benchmarking. In fact, the *seven circles* problem is using as benchmark the solution values at two circles centers, the one dimensional one is using the solution value at the domain midpoint, $x = \pi$, while the *energy decay* one is using the system free energy only. All of the applications compute with `task_time_advance_post-process` the `energy` variable (`Benchmark03` implementation is different from the others), while, for storing the solution values at the points used for benchmarking, `Benchmark02` computes also `u1` and `u2` variables and `Benchmark03` computes `u0`.

5.3 Validation[2]

The implementation of these benchmark applications was instrumental in joining a project whose aim was to provide high accuracy benchmark solutions to Allen–Cahn and Cahn–Hilliard problems in 1D and 2D; to highlight accuracy/efficiency issues of different implementations; and to compare their performance. Several researchers, other than the author, worked on this project together, each one with their own solvers: Zhenlin Guo (University of California, Irvine), Keith Promislow (Michigan State University), Brian Wetton (University of British Columbia), Steven Wise (University of Tennessee Knoxwille), Fengwei Yang (University of Sussex).

The approach here presented, based upon second order five point finite difference stencils for spatial discretization and explicit time stepping, represents the simplest among those adopted. Regular grids with spacing $h = 2\pi/m$ have been used. The adopted Forward Euler (FE) time discretization with fixed time step is first order accurate and conditionally stable; as a consequence, for some of the benchmark problems reported in the following, the level of spatial accuracy required resulted in a maximum stable time step that is too small to be able to perform runs of sufficient simulation time with this method.

Benchmark transition time estimates are determined by linear interpolation between the two computed values on either side of the transition event. The following MATLAB function has been used

```
function T = computeT(fname)

  dat = load(fname);
```

[2]Some parts of the work presented in this section have been published in [20].

	cell–centered				vertex based			
k	m	T	p_k	p_h	m	T	p_k	p_h
$3\ 10^{-2}$	63	48.8126			64	48.7928		
$9\ 10^{-3}$	63	48.7867			64	48.7670		
$3\ 10^{-3}$	63	48.7793	1.00		64	48.7596	1.00	
$9\ 10^{-4}$	63	48.7767	1.00		64	48.7570	1.00	
$3\ 10^{-4}$	63	48.7760	1.00		64	48.7563	1.00	
$9\ 10^{-5}$	63	48.7757	1.00		64	48.7560	1.00	
$9\ 10^{-3}$	127	48.3194			128	48.3171		
$3\ 10^{-3}$	127	48.3122			128	48.3098		
$9\ 10^{-4}$	127	48.3096	1.00		128	48.3073	1.00	
$3\ 10^{-4}$	127	48.3089	1.00		128	48.3066	1.00	
$9\ 10^{-5}$	127	48.3086	1.00		128	48.3063	1.00	
$3\ 10^{-3}$	255	48.2011		2.05	256	48.2008		2.04
$9\ 10^{-4}$	255	48.1985		2.05	256	48.1982		2.04
$3\ 10^{-4}$	255	48.1978	1.00	2.05	256	48.1975	1.00	2.04
$9\ 10^{-5}$	255	48.1976	1.00	2.05	256	48.1973	1.00	2.04
$9\ 10^{-4}$	511	48.1712		2.01	512	48.1712		2.01
$3\ 10^{-4}$	511	48.1705		2.01	512	48.1705		2.01
$9\ 10^{-5}$	511	48.1702	1.00	2.01	512	48.1702	1.00	2.01

Table 5.1: Benchmark I. Computed approximations of the transition time T for $\varepsilon = 0.2$.

```
  t = dat(:,1);
  u = dat(:,2);

  i0=find(u(2:end).*u(1:end-1)<=0);
  if (i0)
    i1=i0+1;
    T=t(i0) - (t(i1)-t(i0)) * u(i0)/(u(i1)-u(i0));
  else
    T=NaN;
  end

end
```

Benchmark I — The number m of mesh cells used to spatially discretize the computational domain in each dimension has been chosen to make the domain center coincide with a computational point: either a cell center or a grid node, depending on the chosen representation (i.e. cell–centered or vertex-based finite differences). For each choice of m, increasingly small time steps have been considered and the corresponding benchmark time computed. From these values it has also been possible to estimate the order of convergence in space and time of this method. Results for $\varepsilon = 0.2$ are reported in Table 5.1.

The order of convergence in space p_h and in time p_k can be estimated using tree subsequent estimates of T. In fact, if T_1, T_2 and T_3 are three estimates computed using the same N and decreasing timesteps, $k_1 > k_2 > k_3$, it is possible to estimate p_k without knowing the exact value of T solving the equation [81]

$$\frac{T_3 - T_2}{r_{23}^p - 1} = r_{12}^p \frac{T_2 - T_1}{r_{12}^p - 1}\,, \tag{5.13}$$

with $p = p_k$ and $r_{ij} = k_j/k_i$.

This equation can be solved explicitly when the ratio between two subsequent discretization parameters is kept constant $r = r_{12} = r_{23}$

$$p = \frac{\log \frac{T_3-T_2}{T_2-T_1}}{\log r}\,.$$

When $r_{12} \neq r_{23}$, equation (5.13) is solved using the following Picard iterative process:

$$p_0 = \frac{\log \frac{T_3-T_2}{T_2-T_1}}{\log r_{12}}\,, \qquad p_{i+1} = \frac{\log \frac{T_3-T_2}{T_2-T_1} - \log \frac{r_{23}^{p_i}-1}{r_{12}^{p_i}-1}}{\log r_{12}} = p_0 - \frac{\log \frac{r_{23}^{p_i}-1}{r_{12}^{p_i}-1}}{\log r_{12}}, \qquad i = 0, 1, \dots\,.$$

Equation (5.13) can also be used to estimate p_h from three subsequent estimates of T computed using the same timestep k while increasing number of grid elements $N_1 < N_2 < N_3$ by setting $p = p_h$ and $r_{ij} = N_i/N_j$.

Since theoretical convergence orders are achieved, it is possible to extrapolate the results in Table 5.1 in m based upon the last two grids and compute T for $m \to \infty$. For example, the vertex-based scheme has a difference of $48.1973 - 48.1702 = 0.0271$. If this is quartered for each subsequent grid level it gives the sequence 0.0068, 0.0017, 0.0004, 0.0001 which yields the extrapolated value of 48.1612. A similar conclusion holds for the cell–centered case.

The equivalent convergence study for smaller choices of ε, based on results reported in Table 5.2 and Table 5.3, gives the following results:

$$T \to 197.71, \qquad \text{for } \varepsilon = 0.1,$$
$$T \to 797.17, \qquad \text{for } \varepsilon = 0.05.$$

The benchmark values extrapolated are in agreement with the values computed by all other solvers in [20] to the second decimal figure, for the cases $\varepsilon = 0.2, 0.1$, and to the first decimal figure, for $\varepsilon = 0.05$. In this latter case agreement to the second decimal figure is achieved only with one of the other solver (labelled Cb in [20]) which implements a multigrid solver for BDF2 finite difference schemes [102].

Benchmark II — The same criterion for choosing both spatial and temporal discretization steps has been used. For this application, however, the stability constraint associated with the explicit time step becomes a practical barrier as m increases, which means that even for $\varepsilon = 0.1$ we have only just started to approach the asymptotic regime that allows us to extrapolate values for T_1 and T_2 in the limit as $m \to \infty$. Smaller values of ε require finer spatial discretization steps

	cell–centered				vertex based			
k	m	T	p_k	p_h	m	T	p_k	p_h
$3\ 10^{-2}$	63	209.4351			64	209.0065		
$9\ 10^{-3}$	63	209.3992			64	208.9705		
$3\ 10^{-3}$	63	209.3889	1.00		64	208.9602	1.00	
$9\ 10^{-4}$	63	209.3853	1.00		64	208.9566	1.00	
$3\ 10^{-4}$	63	209.3842	1.00		64	208.9556	1.00	
$9\ 10^{-5}$	63	209.3839	1.00		64	208.9552	1.00	
$3\ 10^{-2}$	127	200.2524			128	200.2122		
$9\ 10^{-3}$	127	200.2192			128	200.1790		
$3\ 10^{-3}$	127	200.2097	1.00		128	200.1694	1.00	
$9\ 10^{-4}$	127	200.2063	1.00		128	200.1661	1.00	
$3\ 10^{-4}$	127	200.2054	1.00		128	200.1651	1.00	
$9\ 10^{-5}$	127	200.2050	1.00		128	200.1648	1.00	
$9\ 10^{-3}$	255	198.3261		2.25	256	198.3214		2.24
$3\ 10^{-3}$	255	198.3167		2.25	256	198.3120		2.24
$9\ 10^{-4}$	255	198.3134	1.00	2.25	256	198.3087	1.00	2.24
$3\ 10^{-4}$	255	198.3125	1.00	2.25	256	198.3077	1.00	2.24
$9\ 10^{-5}$	255	198.3121	1.00	2.25	256	198.3074	1.00	2.24
$3\ 10^{-3}$	511	197.8629		2.05	512	197.8623		2.05
$9\ 10^{-4}$	511	197.8596		2.05	512	197.8590		2.05
$3\ 10^{-4}$	511	197.8587	1.00	2.05	512	197.8581	1.00	2.05
$9\ 10^{-5}$	511	197.8584	1.00	2.05	512	197.8578	1.00	2.05

Table 5.2: Benchmark I. Computed approximations of the transition time T for $\varepsilon = 0.1$.

	cell–centered				vertex based			
k	m	T	p_k	p_h	m	T	p_k	p_h
$3\ 10^{-2}$	127	844.5886			128	843.6202		
$9\ 10^{-3}$	127	844.5436			128	843.5758		
$3\ 10^{-3}$	127	844.5307	1.00		128	843.5631	1.00	
$9\ 10^{-4}$	127	844.5262	1.00		128	843.5586	1.00	
$3\ 10^{-4}$	127	844.5249	1.00		128	843.5574	1.00	
$9\ 10^{-5}$	127	844.5244	1.00		128	843.5569	1.00	
$3\ 10^{-2}$	255	807.3297			256	807.2487		
$9\ 10^{-3}$	255	807.2890			256	807.2081		
$3\ 10^{-3}$	255	807.2774	1.00		256	807.1965	1.00	
$9\ 10^{-4}$	255	807.2733	1.00		256	807.1924	1.00	
$3\ 10^{-4}$	255	807.2721	1.00		256	807.1913	1.00	
$9\ 10^{-5}$	255	807.2717	1.00		256	807.1909	1.00	
$9\ 10^{-3}$	511	799.7058		2.28	512	799.6963		2.28
$3\ 10^{-3}$	511	799.6943		2.28	512	799.6848		2.28
$9\ 10^{-4}$	511	799.6903	1.00	2.28	512	799.6807	1.00	2.28
$3\ 10^{-4}$	511	799.6892	1.00	2.28	512	799.6796	1.00	2.28
$9\ 10^{-5}$	511	799.6887	1.00	2.28	512	799.6792	1.00	2.28

Table 5.3: Benchmark I. Computed approximations of the transition time T for $\varepsilon = 0.05$.

	cell–centered			vertex based		
k	m	T_1	T_2	m	T_1	T_2
10^{-4}	62	6.6699	25.6796	64	6.6428	26.6155
10^{-5}	62	6.6697	25.6794	64	6.6427	26.6153
10^{-6}	62	6.6697	25.6794	64	6.6427	26.6153
10^{-5}	126	6.3957	26.2164	128	6.3964	26.1783
10^{-6}	126	6.3957	26.2164	128	6.3964	26.1782
10^{-6}	254	6.3509	26.0519	256	6.3508	26.0509

Table 5.4: Benchmark II. Computed approximations of the transition times T_1, T_2 for $\varepsilon = 0.1$.

	cell–centered		vertex based	
k	m	T	m	T
$3\ 10^{-4}$	63	7347.3036	64	7282.6372
$9\ 10^{-5}$	63	7347.2837	64	7282.6174
$3\ 10^{-5}$	63	7347.2779	64	7282.6117

Table 5.5: Benchmark III. Computed approximations of the transition time T_1.

which correspond to even more restrictive choices of time step and are therefore not reported.

Results are shown in Table 5.4 for $\varepsilon = 0.1$. Extrapolation based on second order convergence yields improved estimates of $T_1 \approx 6.34$ and $T_2 \approx 26.01$. These benchmark values are in agreement with the values computed by all other solvers in [20] to the second decimal figure.

Benchmark III — For this application the spatial resolution required to describe accurately the evolution of the field u imposes a time step stability constraint that is simply too restrictive to perform accurate simulation using this explicit scheme. Only simulation with $m = 63$ and $m = 64$ could be performed with the cell–centered and vertex-based schemes respectively and their results are reported in Table 5.5. However, the coarse spatial resolution allows to compute a value for the benchmark time T which has only the same order of magnitude of the values computed by other codes in [20].

This problem is just one example of many problems that require discretization steps too restrictive to be solved with an explicit solver and that justify the implementation of implicit time schemes in the PhaseField component as presented in Chapter 8.

Benchmark IV — For this application, simulations have been performed for $m = 96, 192, 384$ using cell–centered spatial discretizations. One choice of timestep k to ensure numerically stability has been used for each grid size. It was found that only $m = 384$ yielded spatial accuracy to ensure that the solution evolves following the correct energy profile, as shown in Figure 5.3. As reference profile, $\mathscr{E}_*$, the point-wise values available from an accurate approximation have been used and can be found online [101]. The benchmark for this application is the L^1-log error between the computed free energy $\mathscr{E}$ and the reference energy profile $\mathscr{E}_*$ over a logarithmic time

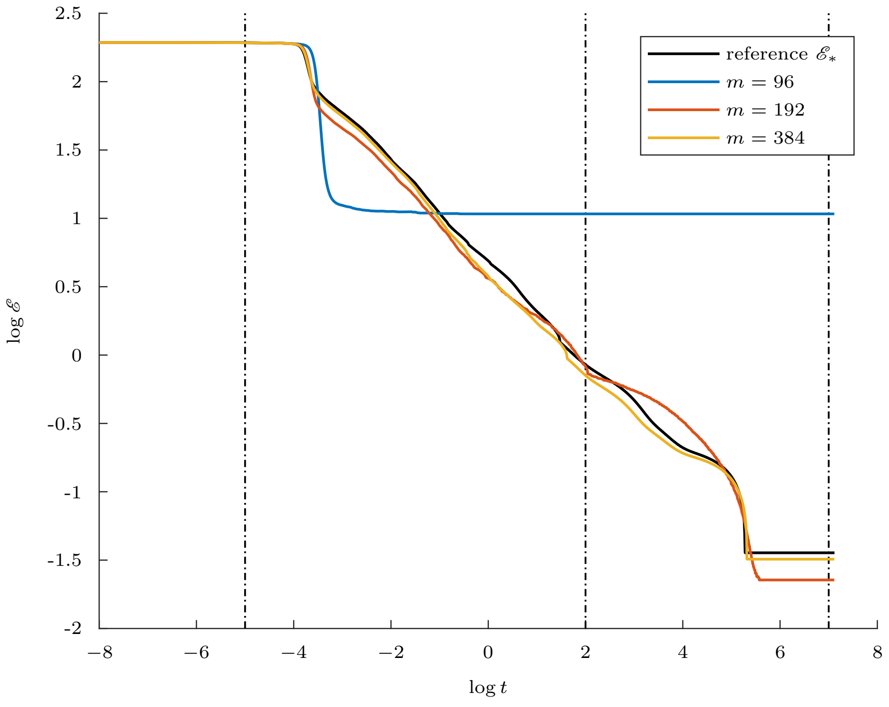


Figure 5.3: Convergence of the energy decay profile for Benchmark IV with respect to grid size m.

span. Specifically, the differences $\mathcal{D}_1$ and $\mathcal{D}_2$ have been chosen as benchmarks

$$\mathcal{D}_1 = \int_{-5}^{7} |\log \mathscr{E}_* - \log \mathscr{E}| \,\mathrm{d}[\log t] \tag{5.14}$$

$$\mathcal{D}_2 = \int_{-5}^{2} |\log \mathscr{E}_* - \log \mathscr{E}| \,\mathrm{d}[\log t] \tag{5.15}$$

These are proposed because often in applications the exact details of the computational results are not important, but the trend of the evolution of length scales is a key feature [5]. The quantity $\mathcal{D}_1$ measures the difference over the full dynamics, while $\mathcal{D}_2$ covers only the first part of the dynamics and omits the fine details of the final transition to steady state.

The convergence in the energy profile as the grid and time step are refined is shown in Table 5.6, where the integrals in (5.14) and (5.15) have been approximated with the trapezoidal rule over one thousand equispaced logarithmic times using linear interpolation of the computed energy profile to estimate the value of $\mathscr{E}$ at such nodes.

Benchmark values for problem I and II are in perfect accordance with the values all solvers agreed on in [20]. In addition, convergence to an accurate energy profile is shown for problem IV. On the other hand, problem III cannot be solved with sufficient accuracy in a feasible time

k	m	$\mathcal{D}_1$	$\mathcal{D}_2$
$4.2\ 10^{-4}$	96	$1.21\ 10^{1}$	$2.70\ 10^{0}$
$5.4\ 10^{-5}$	192	$1.12\ 10^{0}$	$5.06\ 10^{-1}$
$4.8\ 10^{-6}$	384	$6.02\ 10^{-1}$	$3.25\ 10^{-1}$

Table 5.6: Benchmark IV: Convergence of the logarithmic energy profile in $\mathcal{D}_1$ (5.14) and $\mathcal{D}_2$ (5.15).

with the Explicit Euler scheme and cannot be used to benchmark the PhaseField" component. However, all other benchmark problems demonstrated the reliability of the finite differences implementation of spatial derivatives and of the Explicit Euler scheme within the PhaseField component and it was therefore possible to move to the next development step.

Chapter 6

PhaseField Component Boundary Conditions

As the result of the previous stage of development, all implementations merged into the same component: the *PhaseField* component. It groups together different applications sharing the same framework for finite–difference spatial discretization: `Heat`, `PureMetal`, `Benchmark01`, `Benchmark02`, `Benchmark03` and `Benchmark04`.

The goal of the next development phase, described in this section, is to implement, within this general framework, a general strategy to enforce boundary conditions. Such an implementation will need to maintain the same level of abstraction already provided by the framework to offer the application developer a simple high level interface to the complexity of the underlying implementation. A detailed analysis of the chosen strategy is given in §6.1.

In expanding the general framework for handling boundary–conditions, both the implementation of mesh adaptivity and implicit time-stepping are intentionally withheld to focus exclusively on the current task. The experience in the implementations of such techniques for the previous *PhaseField* and *Heat* components will, however, influence the boundary–conditions implementation as reported in §6.2.

To assess the reliability of the implementations performed at this stage, the benchmark problems used at the previous step cannot be used as their domains are periodic. Instead, a convergence analysis for problem (2.25) is performed by comparing the analytical solution (5.2) to the one computed by the *PhaseField* component for the `Heat` application. A second validation is also performed by comparing the values of the asymptotic tip velocity from a set of simulations using the `PureMetal` application with values from literature. These results are available in §6.3.

6.1 Analysis

The implementation of boundary conditions has been already discussed in §4.3.2 for the previous components. In that implementation, the task updating the solution on a given patch was, first, computing the range of internal grid elements and invoking a default version of the appropriate kernel and then, for each boundary face, it was retrieving the kind of boundary condition and the corresponding value to pass as arguments to an overridden version of the kernel for updating the solution. This second version of the kernel featured many nested if statements to switch between implementations and select the appropriate one for updating the solution at the boundary.

In that first step, each patch was partitioned into sub regions over which the implementation selected for time advancing the solution was the same. The resulting sub regions were the same at each time step since the partitioning step depends exclusively on which boundary conditions the user has specified in the problem specification file, on the problem dimension, on the variable basis, and on the given stencil width. Nevertheless, in the old implementation, this step was performed at each time step. To improve the efficiency of the solver it is thus desirable to introduce a structure for saving this partition and avoiding redundant computations. Moreover, the interfaces to both this structure and to the methods used to create it will need to be generic, in order to fit into the general framework and to allow the application developer to implement the same one task for all possible choices of dimension, variable basis and stencil.

The second step of the *Heat* boundary–conditions implementation was to choose the appropriate implementation for updating the solution at each boundary face. This choice was performed, inefficiently, at each time step and for each cell even though, as for the previous step, it could have been performed only once. In order to save the choice made, dynamic polymorphism will be used. Each implementation will be instantiated at the beginning of the simulation and saved into the same structure introduced for the first step. This structure will access these implementations through pointers to a common interface, which is already available in the framework. In fact, all boundary implementations have been introduced to approximate derivatives at boundaries and `FDView` already provides an interface to such functions. It is thanks to the generality of this interface that, to the ultimate advantage for the application developer, it is possible to use the same kernel over each sub region, regardless of whether it is internal or corresponds to a physical boundary.

6.2 Implementation

The implementation of boundary conditions within the *PhaseField* component is presented below in four subsections. The first one describes the design of the `Problem` class template that has been introduced to hold all information and views required to update the solution on each of the subregions identified by partitioning the patches. In the second one, details are given on the boundary views implementation. These class templates provide the high level interface to the different low level implementations for approximating the differential operators which allows the application developer to use the same one kernel for updating the problem variables on all subregions of each patch. The third subsection describes the partitioning algorithm that has been implemented to divide each patch. Its output is a list containing one `Problem` for each subregion. In the last of the four subsections, focus is given to how `Problems` have been used – and could be used by application developers – to enforce user defined boundary conditions in *PhaseField* applications.

6.2.1 The `Problem` Class Template

From the previous analysis emerges the importance of designing the container that represents each one of the subproblems into which the computations performed over each patch are going to be partitioned. The term *subproblem*, or just *problem*, is hereafter used to describe collectively both each one of the regions in the patch partition together with the corresponding view implementation instances and their interfaces. This structure must offer the same interface to all different applications. Since: (1) the choice of the variable basis affects the range of grid indices belonging to a given patch, (2) which faces can be boundary faces depends on the problem dimension (all patches within Uintah are three dimensional), and (3) the thickness of each face depends on the chosen stencil, the `Problem` class must be templated on both `VarType` and `StnType`.

Each subproblem region is rectangular and, so, can be uniquely identified by its `Level`, and the `IntVectors` corresponding to its first and past-the-end indices. Moreover each region is characterized by the list of patch faces it belongs to, with internal regions identified by empty lists. These lists can contain at most as many faces as the problem dimension, with the constraint that any pair of faces in the list cannot be parallel.

In general, each patch can be partitioned into at most 3^d subproblems, but fewer are required if boundary conditions are not applied to some of its faces. For example, a patch for a two dimensional problem with only its $\langle x^- \rangle$- and $\langle y^+ \rangle$-faces lying on physical boundaries (as illustrated in Figure 6.1) must be partitioned into only four regions: the $\langle x^- \rangle$- and $\langle y^+ \rangle$-edges, whose face lists are $\{x^-\}$ and $\{y^+\}$ respectively; the $\langle x^-, y^+ \rangle$-vertex, whose face list is $\{x^-, y^+\}$; and the remaining region, whose face list is empty. Hereafter we will call this remaining region *internal* and, consequently, the corresponding problem, despite it consisting of not only the internal region of a patch but also all its faces that are not lying on physical boundaries. Moreover, it is worth remarking that the same face list may represent different kinds of region depending on the problem dimension: $\{x^-\}$ is a vertex in one dimension and a face in three.

For holding just a partitioned region, there would be no need to make the subproblem container templated on either the `VarType` nor the `StnType` but, since it is also holding instances of `FDView` implementations, it is necessary to templatize the `Problem` class also on these non-type parameters and also on the list of field types of those variables for which boundary condition

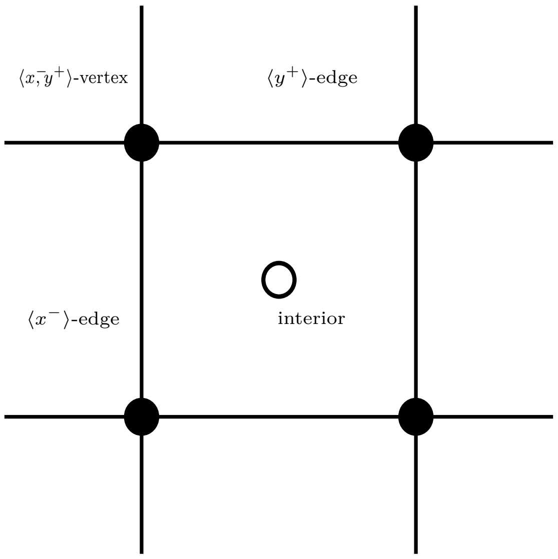

$\langle x^-y^+\rangle$-vertex
$\langle y^+\rangle$-edge
$\langle x^-\rangle$-edge
interior

Figure 6.1: Example of part partitioning of a patch for a two dimensional cell–centered problem when the stencil width is equal to one.

may be imposed. These variables will be referred to as *boundary variable*s. Depending on the particular application, in fact, it may be necessary to impose boundary conditions on multiple variables. For this reason, the subproblem container is designed to specify the list of `Field` type for these boundary variables as a template parameter pack. The new class template is named `Problem` and is declared as follows.

```
template <VarType VAR, StnType STN, typename... Field> class Problem;
```

Two versions each of the `get_view` and `get_fd_view` are implemented for getting a `View` or `FDView` *interface* to a particular view instance. Integer template parameters, are used to specify which boundary view within the `Field...` parameter pack to retrieved and, for vector fields, which one of its components. The first version of each method access view interfaces without retrieving any data from the data-warehouse; the other gets the views and at the same time retrieves the data necessary to evaluate them over the problem's partitioned region. The first version can be used whenever the same view instance may be used over different data-warehouse, for example across multiple timesteps; while the second version is useful when the view instance is used immediately to access the data-warehouse.

The choice of using a parameter pack rather than just a single parameter is driven by the sake of generality for allowing higher dimensional fields, such as tensor fields, in future developments.

Two constructors are available for the `Problem` container: one for instantiating subproblems over internal partitions, and one for boundary problems. The first version of the constructor stores the region and variables identifiers from input into the corresponding fields and creates a new DWFDView instance for each boundary variable, since the data-warehouse view implementation provides the finite–difference approximations appropriate for inner regions.

For the second version of the constructor, which is also storing the region and variables identifiers specified in input to the corresponding fields, the parameter pack expansion of Field... is instead not sufficient to instantiate the correct view implementation for each boundary variable. In fact, as it will be shown hereafter, boundary view implementations require as one of their template parameters the index of the corresponding boundary variable within the Field... parameter pack.

It has to be remarked that all constructors for the Problem class template instantiate all views without retrieving any data from the data-warehouse since these instances are created once at the beginning of any single grid simulations. Data retrieval is delayed until a simulation task triggers it through the get_view and get_fd_view methods.

6.2.2 Boundary Views

Before discussing the details of the algorithm used to partition boundary patches, it is worth analyzing how the implementation of boundary FDViews has been designed.

In order to isolate the enforcement of boundary conditions from the knowledge of the overall geometry, it is important to provide a *local implementation* for each condition. This is an implementation that is not aware either of the geometry of surrounding patches, or of the particular characteristics of the region over which it is being applied to, and that focuses on the imposition of one specific boundary condition over one generic face. These local implementations, detail::bc_fd, are the boundary counterparts of the data-warehouse view implemenetation and, in analogy to them, they implement the methods

```
template <DirType DIR> inline T d ( const IntVector & id ) const;
template <DirType DIR> inline T d2 ( const IntVector & id ) const;
```

for approximating the first and second order spatial derivatives. These local boundary implementation, as with its data-warehouse counterpart, are parameterized upon the variable field type, the stencil and variable basis as well as the width of the stencil, and, in addition to them, also on the face and boundary condition types. The face parameter is used to deduce, at compile time, static constant expressions for the direction and sign of the outward vector normal to the given face. At this stage, implementations are given for Dirichlet and Neumann boundary conditions for stencils whose width is equal to one. In particular, the same implementation is used for the P3, P5, and P7 stencils. Additional implementations can be introduced for new boundary condition types by defining new partial specializations of the detail::bc_fd template for newly defined values of the BC enumeration or, for new stencils, by defining new partial specializations of the template interface for newly defined values of the StnType enumeration and its corresponding width.

In order to be able to evaluate their variables over their boundary region, each detail::bc_ fd instance stores a pointer to the detail::view interface which points to the view implementation instance that has been passed to their constructor. Even if not strictly required by the currently implemented stencils, the use of the general detail::view interface is ensuring that these boundary implementations are local by taking away the need for any knowledge of the surrounding geometry. It may happen, for example, that for approximating the derivative along the x direction at a boundary vertex, say the $\langle x^-, y^-\rangle$-vertex, with a wider stencil it is required to evaluate the variable field also at grid entries that are on the other side of the $\langle y^-\rangle$-face; for guaranteeing its local nature the implementation cannot be aware of the fact that the patch $\langle y^-\rangle$-face lies on a physical boundary, and so it will try to retrieve that value that is out of the computational domain hrough its inner view interface. The actual implementation of how to extrapolate the field value at that ghost point is, in this setting, hidden behind that interface. The only ghost elements that a detail::bc_fd specialization has to account for are those lying on the other side of the boundary face upon which they are parametrized.

The same strategies presented in §2.3.2, have been adopted for the d and d2 methods.

The set view virtual method is provided for setting view supports which, in this case, are boundary regions and never coincides with a whole patch. Therefore, the version of the set methods for setting the support over a patch is implemented to produce a runtime error. The only fully implemented version of the set method uses the given boundary region to compute the bounding indices of the ghost region on the other side of the view boundary face, which will be its support. This region can be accessed by the get_support method and it is possible to check if an index belongs to it using the is_defined_at member function. The square brackets operator operator[] is implemented, in its constant version, to return the value of its field extrapolated over a ghost index, while its non–constant version raises a runtime error (being meaningless to assign the field value at a ghost index).

The remaining methods required to fully implement the virtual detail::basic_fd_view interface are implemented, rather than in each detail::bc_fd specialization, within the ScalarField specialization of the detail::bc_basic_fd_view template interface since their implementation is independent of the actual boundary conditions.

The detail::bc_basic_fd_view template interface is templated differently from the detail::bc_fd implementation to better resemble those of the detail::basic_fd_view abstract interface: in addition to the detail::basic_fd_view parameters, field and stencil type, and to the variable basis of the inner view, a non-type template parameter of type BCF is used. This parameter merges together, into a single integral value, the boundary face and condition types. The packed value of the different boundary enumerators' values – which will be referred to as *boundary pack* – is computed, and unpacked, using bitwise operations taking advantage of the fact that the values of the scoped and typed BC enumeration have non-zero bits only within the second byte and that those of the FaceType one are using just the first byte.

The so implemented detail::bc_basic_fd_view class, despite being a full implementation of the detail::basic_fd_view, is not suitable for replacing the dw_basic_fd_view on boundary regions. In fact, it has no control on the view used to access the data-warehouse and, since it can be used to compute derivatives only along the normal to its boundary face and to evaluate the field over its ghost region, it cannot handle boundary regions with co-dimension grater than one (because of the use of three–dimensional patches regardless of the problem dimension these regions are always patch faces, even when they represent edges or vertices). The boundary equivalent of the `detail::dw_basic_fd_view` template interface is indeed named `detail::bcs_basic_fd_view`, where the additional `s` is used to specify that this class can handle regions over which multiple boundary faces intersect and multiple boundary conditions apply.

The idea for handling boundary regions, – faces, edges or vertices – in a generic fashion is to develop an implementation of the `detail::basic_fd_view` which provides an efficient mechanism to switch between multiple view instances: one `detail::dw_basic_fd_view` instance to access data, and as many `detail::bc_basic_fd_view` instances as there are boundary faces intersecting over the given boundary region for extrapolating field values at ghost regions and approximating derivatives.

For this reason the `detail::bcs_basic_fd_view` template interface is parametrized, other than on the field and stencil type, also on the parameter pack `P...` providing the list of boundary packs over a boundary region. The introduction of the boundary pack, `BCF`, allows the use of different boundary conditions for each face.

Another two template parameters are used to characterize the `detail::bcs_basic_fd_view` template interface: `Problem` and `Index`. The `Problem` typename is used to specify a particular *type instance* of the `Problem` class template introduced in the previous subsection. The `Index` typename, instead, specifies which position within the `Problem`'s list of boundary variables corresponds to the particular instance of `detail::bcs_basic_fd_view`. `Index` can also indicate, for vector fields, which component of the specified boundary view the `detail::bcs_basic_fd_view` type instance corresponds to.

Of the two types of constructors usually provided for view implementations, only the one that defer the retrieving of data from the data-warehouse has been implemented, since boundary views are designed to be instantiated only at the beginning of a simulation and to then access the data-warehouse at each time step. It is templated on the variable field type and contains a `bc` field, for the `BC` enumerator corresponding to the type of boundary condition in the input file, and a `value` field, for its boundary value. This may have the same type as the variable field value type if it is a `ScalarField`, or a `array` with the same type and size of the variable field components if this is a `VectorField`.

The `detail::bcs_basic_fd_view` class template scalar implementation derives from the `detail::piecewise_view` class template. This class has been introduced to wrap together several `detail::view` interfaces into a unified `detail::view`. As with the `detail::view` interface, it is templated on the field type. When evaluating a `piecewise_view` at a location, a loop over its view is performed to find the first view that is defined over that location and the value of this inner view is returned as the valued of the `piecewise_view` at such location.

The `detail::piecewise_view` class template, as with the `detail::bcs_basic_fd_view`, is designed to be used only as a base class for other implementations of the view interface and leaves the management of the inner view list to the super classes. The inner view list is declared to be a protected member and it is this list that is being populated in the body of the `detail::bcs_basic_fd_view` constructor listed above.

Before moving to the description on how these lists of `Problem` instances are created, the `BCFDView` class template is still to be introduced. This is the boundary implementation of the `FDView` interface to be used by application developers.

The `BCFDView` class is templated only on the problem type, the field index and the list of boundary packs. The template parameters required by its base class

Lastly, the `BCFDView` class template also inherits from the `Implementation` type instance for the `FDView` interface. This provides, using the factory design pattern, an encapsulated method for the dynamic instantiation of boundary implementation of the polymorphic `FDView` interface. As explained in §5.2, in order to be able to use a particular type instance of the `BCFDView` template, it is necessary to define its name and register it to its factory. For each instantiated `FDView` type a `Factory` type instance is required, and the `BCFactoryFDView` alias template can be used to specify these with the same template parameter list used for their `FDView` base interface. For convenience, alias templates are defined for the `Problem` class templates required by those applications introduced so far in the *PhaseField* component for which boundary conditions can be given. The physical model of the `Heat` application has only one scalar variable, the temperature field, which is also a boundary variable, and so the following `Problem` alias typename is defined for it:

```
template<VarType VAR, StnType STN>
using HeatProblem = Problem < VAR, STN, ScalarField<const double> >;
```

The `PureMetal` application, instead, has not only two model variables – the phase and the temperature scalar fields – which are both boundary variables, but also artificial variables which are also differentiated in the model and, therefore, require to be evaluated over ghost regions which is implemented by imposing boundary conditions. These are the square of the anisotropy function scalar field, and the vector field containing the terms arising from the spatial differentiation of the anisotropic function, whose size is dependent on the problem dimension. The corresponding `Problem` alias template definition is listed below.

```
template<VarType VAR, StnType STN>
using PureMetalProblem = Problem < VAR, STN,
    ScalarField<const double>, ScalarField<const double>, ScalarField<const double>,
    VectorField < const double, combinations< get_stn<STN>::dim, 2>::value > >;
```

The number of type instances of the `BCFDView` template required by an application, given its dimension d, can be computed as the product of the number of variable bases available (2), the number of available stencils (in the current implementation 1), the number of boundary

Problem	DimType	# VarTypes	# StnTypes	# Indexes	# P...	# BCFDView
HeatProblem	D2	2	1	1	24	48
HeatProblem	D3	2	1	1	124	248
PureMetalProblem	D2	2	1	4	24	192
PureMetalProblem	D3	2	1	4	124	992

Table 6.1: Number of `BCFDView` type instances required by phase field applications.

variables for that application (1 for heat and 4 for pure metal) and the number of possible combinations of boundary faces/conditions[1] $(5^d - 1)$. In Table 6.1 such numbers are listed for all phase field application except for benchmark applications, as they are defined on periodic domains and, thus, do not have any boundary condition.

It would be unpractical, and error prone, to write the source file with the definitions of the name for all these type instances without using a script. For this reason, the *BCFDView-bld.sh* bash script has been implemented.

The diagram of all classes introduced for boundary views is given in [18] together with a more detailed description of their implementations.

6.2.3 Partition Algorithm

The review of the implementation of boundary views is now complete and it is left to analyze the algorithm used to partition the problem of updating the solution over each patch into subproblems. They consist of both the partitioned region and of the views required to get variables' grid values and evaluate spatial derivative approximations.

The partitioning is performed in two steps. The first of which is to scan the list of boundary conditions provided by the user in the Uintah Problem Specification file and identify, for each patch, which faces are *boundary faces*, which are those faces that lie on the domain boundary and over which a boundary condition has been specified for at least one of the given set of boundary variables. Independently from the problem dimension, d, grids are always three–dimensional in Uintah, so, for lower dimensional problems, the computational domain is shaped to have only one grid cell along directions whose (zero based) index is less that d. When looking for boundary conditions, then, only faces with normal vectors parallel to the first d directions are to be considered and so, since on a patch there are two faces per direction, an array of flags with twice as many entries as the problem dimension is used to keep track of which faces are boundary faces. In this first step, all information about boundary conditions is organized in one array of `BCInfos` per boundary variable with the same size of the flag array. In this way boundary information for a given variable on a given face can be easily accessed by retrieving the face value as the index corresponding to that variable in the array.

The second step is to partition patches into subregions and instantiate views of the appropriate type for each boundary variable on each partition of each patch. This is performed

[1]For each of the d allowed directions, five cases are possible (Dirichlet condition on either the positive or the negative face, Neumann condition in either face, or no conditions) except for the combination where no condition is given in any directions.

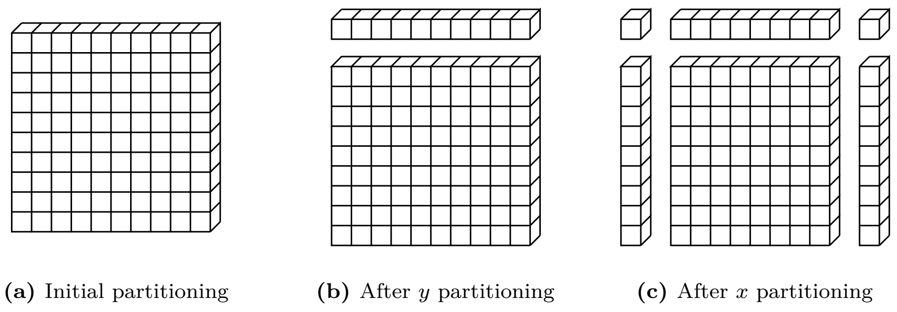

(a) Initial partitioning **(b)** After y partitioning **(c)** After x partitioning

Figure 6.2: Example of computed subregions in the problem partitioning of a patch for a 2D problem with boundary faces at $\langle x^-\rangle$-, $\langle x^+\rangle$- and $\langle y^+\rangle$-faces.

using the flags and boundary information arrays from the previous step. The set of partitioned regions is initially set to contain the whole patch and, starting form the highest direction less than d, recursion is used to split each region in the set of partitioned regions accordingly to the patch boundary flags for that direction, until the lowest direction, `X`, is reached. A graphical representation of this second step is given in Figure 6.2. Views for each boundary variable are then instantiated and the partitioning is complete.

This algorithm is implemented, within the `BCInterface` stucture template, as the `partition_patch` static template method. The list of partitioned subproblems is returned by this method as a `list` of `Problems` with, as template parameters, the same values of variable basis and stencil type of its `BCInterface` container template parameters and the same list of field types used for the method itself.

The class diagram of the `BCInterface` and the functors introduced to implement its `partition_patch` method is given in [18].

6.2.4 Applications: How to Enforce Boundary Conditions

Thanks to the introduction into the general framework of the subproblem partitioning and of the boundary views, only few modifications have been necessary to handle generic boundary conditions in *PhaseField* applications. The following description of the modifications made to the `Heat` application is the best example of how much simpler it is for the application developer to implement boundary conditions within the application when compared to their previous implementation in §4.3.

In the `Heat` application, the only boundary variable is the temperature field, u. The alias template `HeatProblem`, has already been introduced for being used instead of `Problem` template in the development of this component. This alias is parameterized on the application variable basis and stencil type and is equivalent to the `Problem` template where the variadic list of boundary field types is composed by only the type for the temperature field, `ScalarField<const double>`. For convenience the static constant expression `U` has been defined within the `Heat` class to be used as the index in the boundary view list corresponding to the temperature field. In addition to this, only the `subproblems_label` has been added to the application members for storing the pointer to the `VarLabel` object used to identify subproblems in the data-warehouse. To instantiate and to free the instances of this label together with the application, a line has been added to both the `Heat` constructor and destructor for calling, respectively, the `VarLabel` constructor and destructor. A line has been added also to the `problemSetup` method for telling the scheduler how to behave with the subproblem variable (currently Uintah, due to their generality, leaves to application developer the responsibility on how `PerPatch` variables are updated and moved between data-warehouses and this line is required to tell the scheduler that management of the subproblem variables is completely handled by the application).

Bigger changes are required for the application scheduler members. An additional task has to be scheduled by the `scheduleInitialize` method to initialize the subproblems variable at the beginning of the simulation. For isolating the scheduling of this task from the one initializing the solution variable two separate scheduling private methods are defined: `scheduleInitialize_subproblems` and HeatscheduleInitialize˙solution. These private methods are called in sequence when the public `scheduleInitialize` method is invoked. The first of the two private schedulings tells the scheduler that for each patch the `task_initialize_subproblems` can be used to create the subproblems variable on each patch with no other variable required. The body of second the one is unchanged from the original body of the previous `scheduleInitialize` implementation since the same `task_initialize_solution` task as before can be used to initialize the temperature field variable. The `task_initialize_subproblems` implementation is very simple: it loops over the given `PatchSubset` to instantiate, for each patch therein, a `PerPatch` variable and set its data to be a new `SubProblems` object and puts it in the new data-warehouse. This automatically populates the list of subproblems for each patch contained in each `SubProblems` instance.

The other scheduling method that has been modified is the `scheduleTimeAdvance` which now calls the `scheduleTimeAdvance_subproblems` before calling the same private time advance schedulers as before. The `scheduleTimeAdvance_subproblems` is used to add to the `task_time_advance_subproblems` to the queue, which moves the `SubProblems` instances for each patch from the old data-warehouse to the new one. This task is not performing any computation over grid elements and therefore is not using any kernel. It simply gets the subproblems variable instance for each patch from the old data-warehouse and puts it in the new one. All previous private time advance scheduling is also modified to add to their task dependencies the subproblems' variable from the old data-warehouse.

The previous implementations of these tasks all followed the same scheme:

– loop over the given `PatchSubset` and for each patch

    1. instantiate required `DWViews` and `DWFDViews`,
    2. use the `parallel_for` adaptor to execute the corresponding kernel over each grid element in the patch.

In the new implementations, explicit instantiations of DWFDViews are replaced by retrieving FDView interfaces to the view instances already instantiated in the subproblems variable. The new scheme is therefore the following:

– loop over the given PatchSubset and for each patch

  1. retrieve for each patch its subproblems,
  2. instantiate required DWViews,
  3. loop over the retrieved subproblems and for each Problem therein
     (a) get the required FDViews interfaces with its get_fd_view method specifying from which data-warehouse its data has to be retrieved,
     (b) use the parallel_for adaptor to execution the corresponding kernel over each grid element in the problem region.

The body of the patch loop of the task_time_advance_subproblems task, whose previous version is listed in §5.2, is now the following:

```
DWView < ScalarField<double>, VAR, DIM > u_new ( dw_new, u_label, material, patch );

Variable < PP, SubProblems < HeatProblem<VAR, STN> > > subproblems;
dw_new->get ( subproblems, subproblems_label, material, patch );
auto problems = subproblems.get().get_rep();

for ( const auto & p : *problems )
{
    FDView < ScalarField<const double>, STN > & u_old = p.template get_fd_view<U> ( dw_old );
    parallel_for ( p.get_range(), [patch, &u_old, &u_new, this] ( int i, int j, int k )->void {
        time_advance_solution_forward_euler ( {i, j, k}, u_old, u_new ); } );
}
```

No further modifications are needed since boundary conditions implementations are hidden behind the FDView interface and all kernels remain unaltered.

Analogous modifications are made to the PureMetal application to support boundary conditions. The ease with which it is possible to introduce boundary condition in an application is not the only improvement coming from their implementation in this development step. The resulting code is also more efficient since all nested if and switch statements and for loops that were, previously, performed at each time step to select at runtime the correct implementation over a patch are now replaced by a single for loop and the virtual tables used in the resolution of the virtual methods of the FDView interface.

6.3 Validation

Two sets of simulations have been performed to assess the reliability of the implementation of boundary conditions here introduced. The first set of simulations is performed to analyze the convergence to the analytical solution of the Heat problem with Dirichlet boundary conditions as both the space and time discretization steps are reduced. The second set of simulations is performed using the `PureMetal` application. Different post-processing algorithms are used to compute the asymptotic tip velocity of problems with different parameters. These values are then compared to the results from [49].

6.3.1 Convergence Analysis for the Heat Application

As described in §5.2, the solution to the heat problem (2.25) with homogeneous Dirichlet conditions and initial conditions (5.1) over the domain $\Omega = [-L, L]^d$ admits an open form solution (5.2). This allows to evaluate at every timestep a global discrete error of the solution (5.7) and of its first and second order derivatives as (5.11). These errors are therefore computed for different values of the grid step, h, and time step, k, and the speed with which these errors are converging to zero is compared to the order of the truncation error that is computed a priori.

The truncation error, E_t at the $\boldsymbol{j}$-th grid location and timestep t^n is the residual in the discrete model when the discrete solution is replaced with the exact solution u:

$$[D_t^+ u](\mathbf{x}_{\boldsymbol{j}}, t^n) = \alpha \sum_{i=0}^{d-1} [D_\nu D_\nu u](\mathbf{x}_{\boldsymbol{j}}, t^n) + E_t(\mathbf{x}_{\boldsymbol{j}}, t^n),$$

where notations (2.26d) and (2.26e) have been used. Since u satisfies (2.25) we can write, omitting the function arguments:

$$\begin{aligned} E_t &= [D_t^+ u - \alpha \sum_{i=0}^{d-1} D_i D_i u - u_t + \alpha \sum_{i=0}^{d-1} u_{ii} \\ &= \left[D_t^+ u - u_t\right] - \alpha \sum_{i=0}^{d-1} \left[D_i D_i u - u_{ii}\right] . \end{aligned}$$

The the truncation is therefore the sum of two terms: a temporal error which arises from the discretisation of the time derivative and a spatial error due to the discretization of the laplacian operator. These terms can be evaluated using the leading-order error formulae for finite differences approximations (2.27)

$$E_t = \frac{1}{2} u_{tt}\, k + \mathcal{O}(k^2) - \frac{\alpha}{12} \sum_{\nu=0}^{d-1} u_{iiii} h^2 + \mathcal{O}(h^4)$$

$$= \left[\frac{d^2\alpha^2c^4}{2} k - \frac{d\alpha c^4}{12} h^2 \right] u + \mathcal{O}(h^4 + k^2)$$
$$= \frac{d\alpha c^4}{12} \left[6d\alpha\, k - h^2 \right] u + \mathcal{O}(h^4 + k^2)\,.$$

The truncation error for the considered heat problem is therefore linear in time and quadratic in space. Moreover, if $h^2 > 6d\alpha\, k$, the spatial term dominates the temporal one.

For all simulations performed, the same diffusion coefficient has been used ($\alpha = 0.1$). The values for the spatial and temporal discretization steps considered are $h = 2^{-i}$ with $(i = 0, 1, \ldots, 4)$ and $k = 2^{-j}$ with $(j = 1, 2, \ldots, 6)$.

As discussed in §2.3.2 the forward Euler scheme is conditionally stable and the following condition must be verified for the round-off error to not be amplified from one timestep to the next

$$h^2 \geq 2d\alpha\, k\,.$$

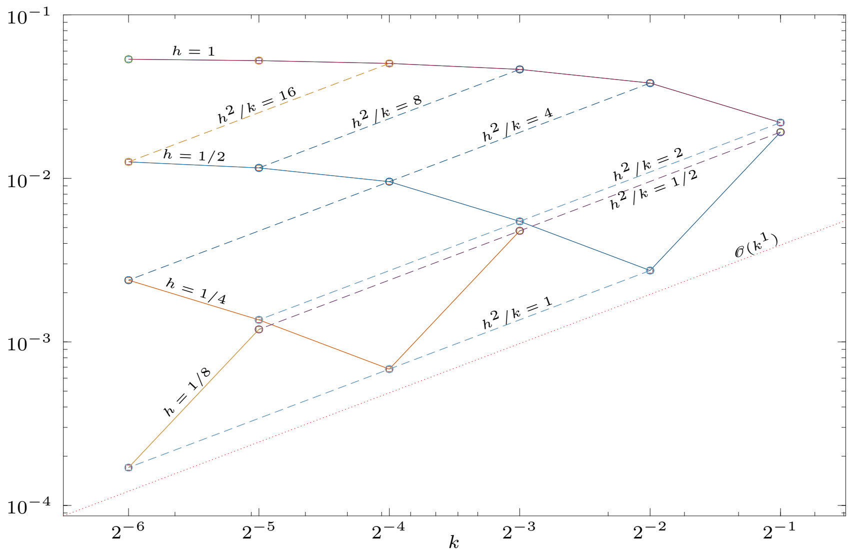
Simulations for which the condition is not satisfied are, in fact, found to be numerically unstable and are not shown in the plots hereafter.

In Figures 6.3–6.4 the discrete 1-norm, $\sum_N |\, \mathbf{err}_p^N\, |$, of the computed global discrete errors (5.11) are plotted in logarithmic scale, first, against the timestep and, then, against the grid step. Values of error corresponding to the same choice of the other discretisation parameter are joint with solid lines, while values with the same $\frac{h^2}{k}$ ratio are joint with dashed lines.

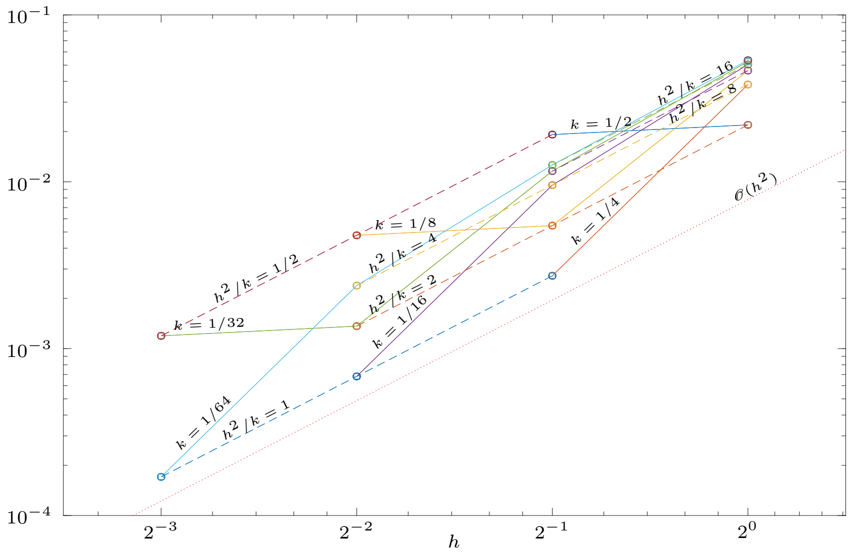
Looking at the solid line in Figure 6.3a, it is evident how halving the timestep is beneficial only for the first stable choice of k. Moreover, the inclination of the dashed lines in Figure 6.3a is the same of the reference line and confirms the expected linear convergence of the error with respect to k. The expected second order convergence in time of the global L^2-discrete error is also evident in Figure 6.3b.

Generally, for solutions with minimal regularity ($\mathscr{C}^1$), one order of convergence in time is lost when the global H_0^1-discrete error is considered. This is expected, therefore to vanish linearly with h, and to be constant in k. In fact, this error is a measure of the discretisation error of the solution's spatial first order derivatives, and is necessarily independent of k. Figure 6.4 confirms that the global H_0^1-discrete error is purely spatial, but a higher quadratic h-convergence is observed. This behavior can be explained by the regularity of the solution (5.2).

Another order of convergence is expected to be lost for the H_0^2-discrete error with $\mathscr{C}^2$ solutions. This will imply that the guaranteed theoretical convergence in both h and k is zero. Being the global H_0^2-discrete error a measure of the discretisation error of the solution's spatial second order derivatives, as for global H_0^1-discrete error, this is found from Figure 6.5 to be constant in k and quadratic in h. In fact, not only the solution (5.2) is analytical, but its second order derivatives are proportional to the solution itself.

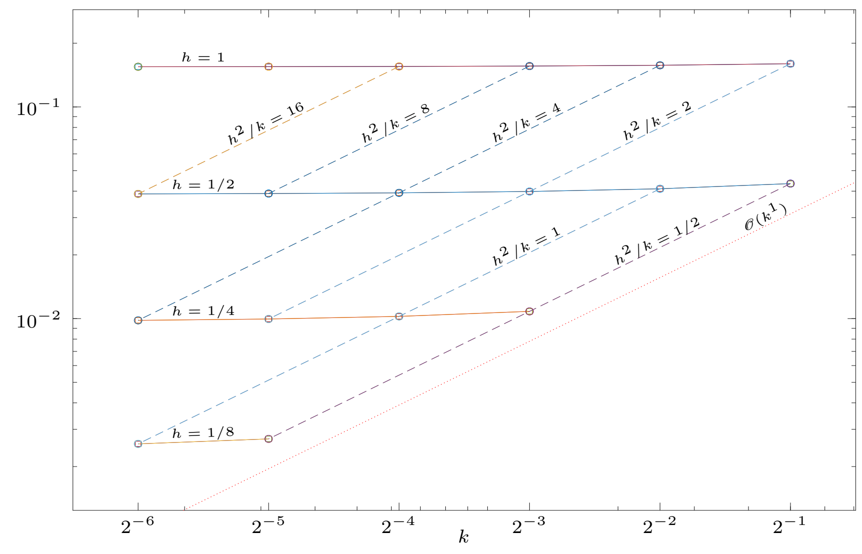


(a) k-convergence

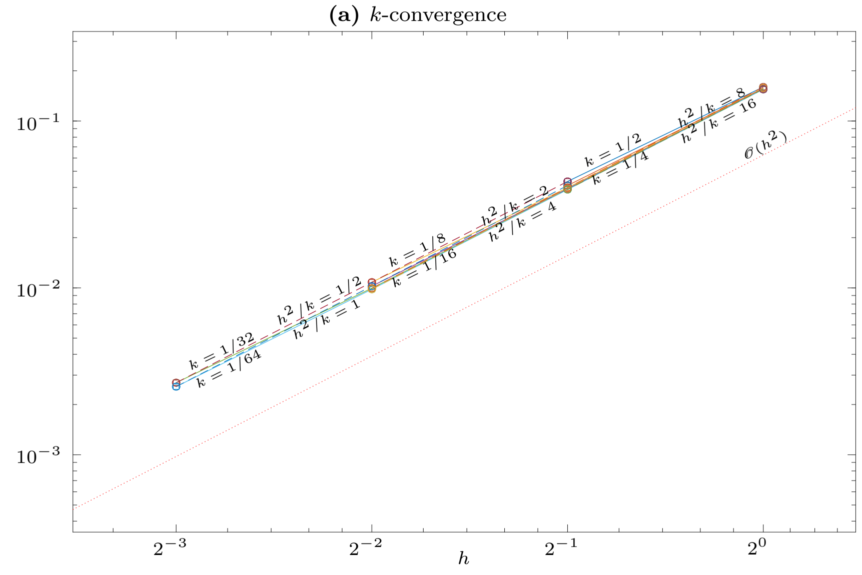


(b) h-convergence

Figure 6.3: Convergence analysis of the global L^2 error for the Heat problem ($\alpha = 0.1$). Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (a) and on the grid size (b). Theoretical order of convergence is plotted for reference as a dotted line.

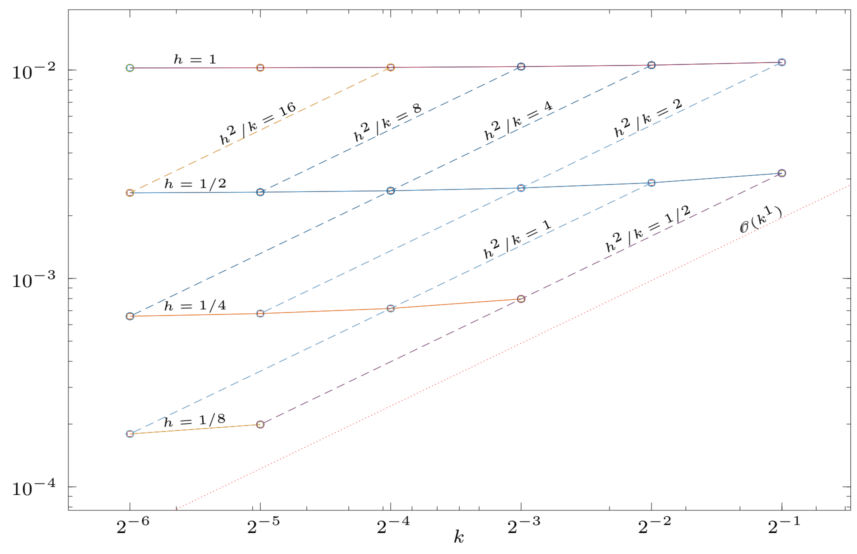


(a) k-convergence

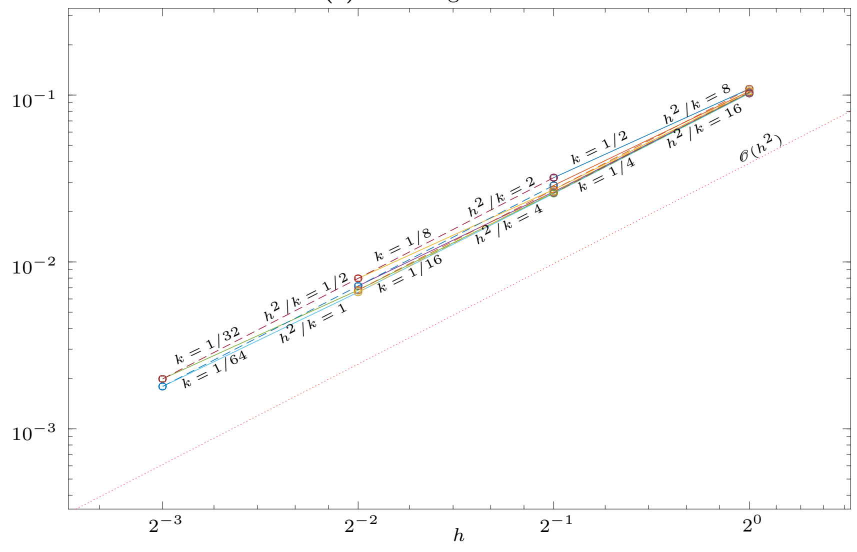


(b) h-convergence

Figure 6.4: Convergence analysis of the global H_0^1 error for the Heat problem ($\alpha = 0.1$). Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (a) and on the grid size (b). Graphs proportional to k^1 and h^2 are plotted for reference as a dotted lines.



(a) k-convergence



(b) h-convergence

Figure 6.5: Convergence analysis of the global H_0^2 error for the Heat problem ($\alpha = 0.1$). Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (a) and on the grid size (b). Graphs proportional to k^1 and h^2 are plotted for reference as a dotted lines.

	Δ	$\lvert\varepsilon\rvert$	D	d_0	$\tilde{V}_{\text{tip}}$	V_{tip}
I	0.65	0.05	1	0.554	0.0469	0.0847
II	0.55	0.05	2	0.277	0.0170	0.1227
III	0.60	0.03	2	0.277	0.0188	0.1357
IV	0.55	0.02	2	0.277	0.00685	0.04946

Table 6.2: Steady-state velocity obtained by Green's function calculations for different sets of problem parameters.

6.3.2 Asymptotic Tip Velocities for the PureMetal Application

The implementation of boundary conditions, made possible to run simulations using the PureMetal application of the PhaseField Uintah's component. To benchmark the implementation it is possible to compare quantitatively the asymptotic tip velocity computed from the solution when a constant value is reached with its steady-state value computed by means of the Green's function method [55, 67]. The input parameteres for those calulations used in [49] to solve the free-boundary formulation (2.8) with $\beta(\varphi) = 0$, and $d_0(\varphi) = d_0(1 - 15\varepsilon \cos 4\varphi)$ with $d_0 = WI/\lambda J$. These input parameters for the Green's function calculations have been chosen to correspond exactly to the anisotropic model (2.3). This model, in fact, reduces in the sharp-interface limit to the free-boundary formulation (2.8) with:

$$d_0(\varphi) = \frac{I}{\lambda J}[W(\varphi) + W''(\varphi)]$$
$$\beta(\varphi) = \frac{I}{\lambda J}\frac{\tau(\varphi)}{W(\varphi)}\left[1 - \lambda\frac{W(\varphi)^2}{2D\tau(\varphi)}\frac{K + JF}{I}\right],$$

where the term in square brackets on the right-hand-side of the equation for β represent the correction arising from the variation of u in the interface thickness.

In [49], the authors provide a table of values for the steady-state velocity obtained by Green's function calculations for different sets of problem parameters. These velocity, $\tilde{V}_{\text{tip}}$, are reported dimensionless thanks to the following non-dimensionaliation

$$\tilde{V}_{\text{tip}} = \frac{d_0}{D} V_{\text{tip}} ,$$

where d_0 is the capillary length and D is the thermal diffusivity.

The same discretization parameters ($h = 0.4$, $k = 0.016$) used by the authors have been used here and the explicit time scheme is found to be numerically stable only for four of the eleven set of parameters provided in [49]. These are named with roman ordinals and are listed together with the corresponding non-dimensional and dimensional Green's function steady-state velocities in Table 6.2.

Plots for each problem of the tip position, velocity and curvature together with the arm curvature are shown in Figures 6.6–6.9. In these plots, both results from simulation where the dendrite growth is favored along the Cartesian axes (parallel case) and where the favored growth direction is along the quadrant bisectors are shown. In each case both the polynomial and the

	L_{par}	L_{diag}	$t_{\max}$
I	240	180	2100
II	320	240	2100
III	240	180	1500
IV	320	240	4000

Table 6.3: Width of the computational domain for the parallel and diagonal cases, and maximum simulation time for each set of problem parameters.

hyperbolic tangent post-processing approach are used with either cell centered or vertex based discretization.

Simulations have been performed on the first quadrant using homogeneous Neumann conditions on the lower boundaries while Dirichlet condition are used on the upper boundaries so that the temperature field is there equal to the undercooling, Δ, throughout the simulation. This choice of domanin and boundary conditions on the lower boundaries allows to quarter the problem domain taking advantage of the problem symmetry with respect to the Cartesian axes. Dirichlet conditions are insted modelling a problem where undercooled metal is found all around the computational domain with the assumption that the upper boundaries are far enough from the seed that any change in the temperature and phase field is observable there during the period of time over which the simulation spans. The width of the computational domain for the parallel and diagonal cases, $\Omega = [0, L]^2$, and the maximum simulation time $t_{\max}$, for each set of problem parameters are shown in table Table 6.3.

For Problem I, the effects of grid anisotropy due to the size of h are evident in the plots of the tip positions for all of the parameter sets. Looking at Figure 6.6a, in fact it can be observed that the dendrite is growing faster in the diagonal case than in the parallel case.

From Figure 6.6b it can be observed that the asymptotic tip velocity in Problem I is reached after 1700 time units in the diagonal case and after 1900 time units in the parallel case which is growing more slowly. In the first case the computed velocity is 0.0851 while in the latter case it is 0.0750. The higher error with respect to the green function value is found the parallel case (11%) while very good agreement with the steady-state value is observed in the diagonal case with an error of less than 0.5%. In the diagonal case. moreover, the tip velocity is observed to increase after 1800 time units. This behavior has to be ascribed to the Dirichlet condition imposed at the upper boundary. At this timetep the tip is about 182 spatial units from the origin which is less than 50 units from the computational upper boundary, being the computational domain used for this simulation $[0, 180]^2$. In the parallel case, no boundary effect is appreciable with the chosen computational domain $[0, 240]^2$ with the tip at the final simulation time $t = 2100$ 190 spatial units from the origin which is, evidently, far enough from the upper boundary for not being affected by its Dirichlet condition.

From the curvature plots, Figures 6.6c, 6.9d, it can be only observed that the choice of the variable basis affects the curvatures' computations in the parallel case while the choice of the post-processing algorithm affects the computations in the diagonal case. These figures are however given for completeness and are not compared with any reference value which is not

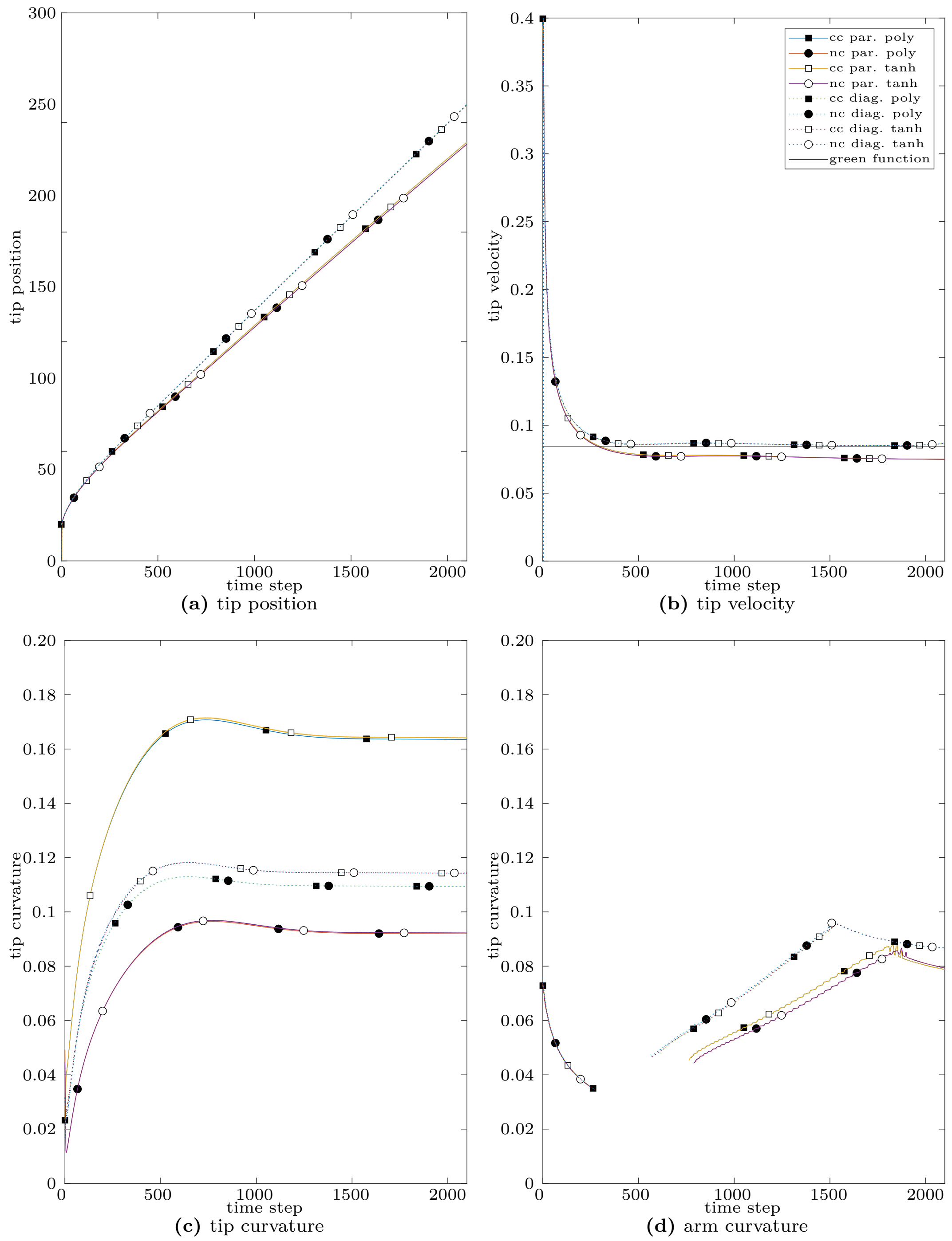
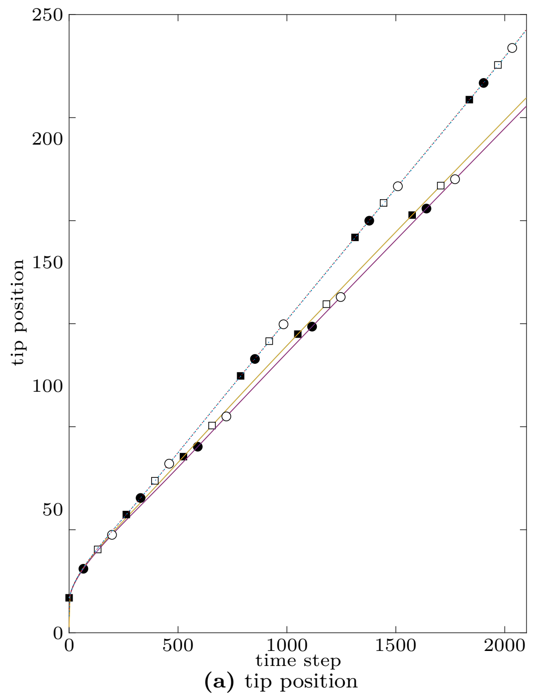
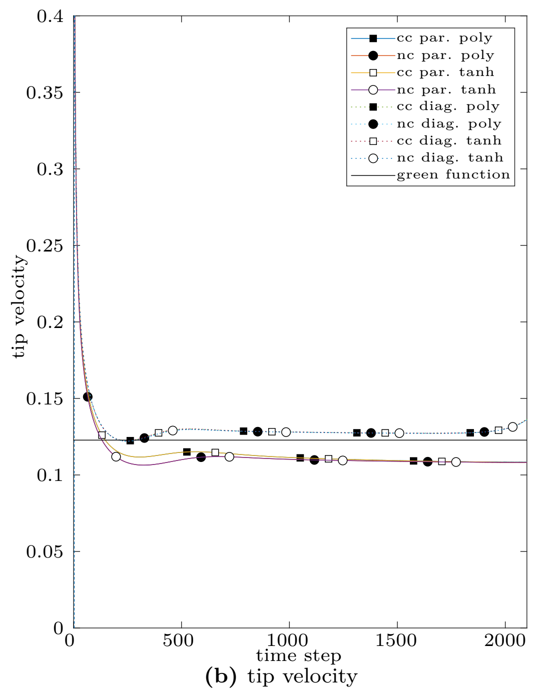


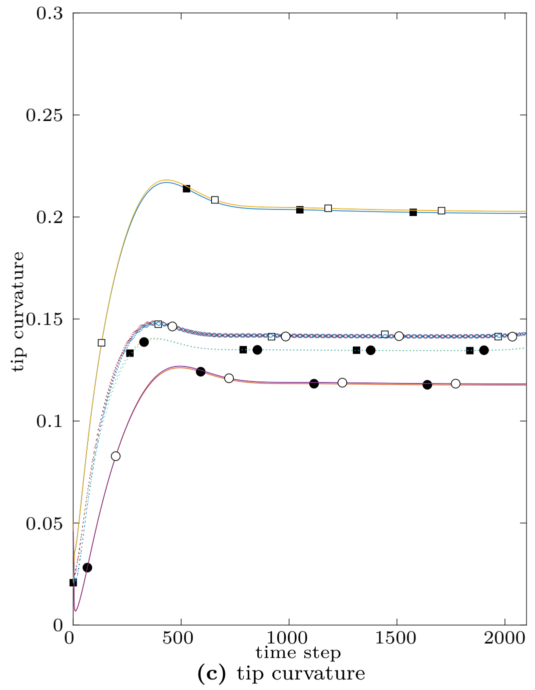
Figure 6.6: Dendrite characterization from the PureMetal simulation using the Problem I set of parameters. Diagonal and parallel growth cases are compared using different post-processing algorithms. Steady-state velocity computed with the green function method is shown for reference in (b).



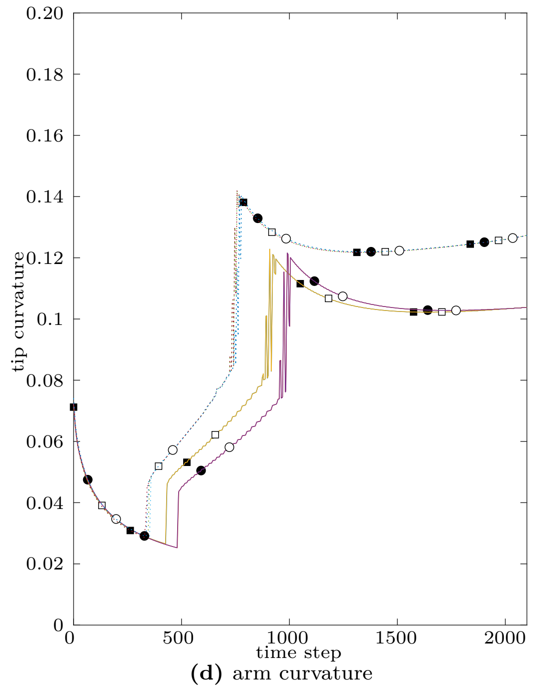
(a) tip position



(b) tip velocity

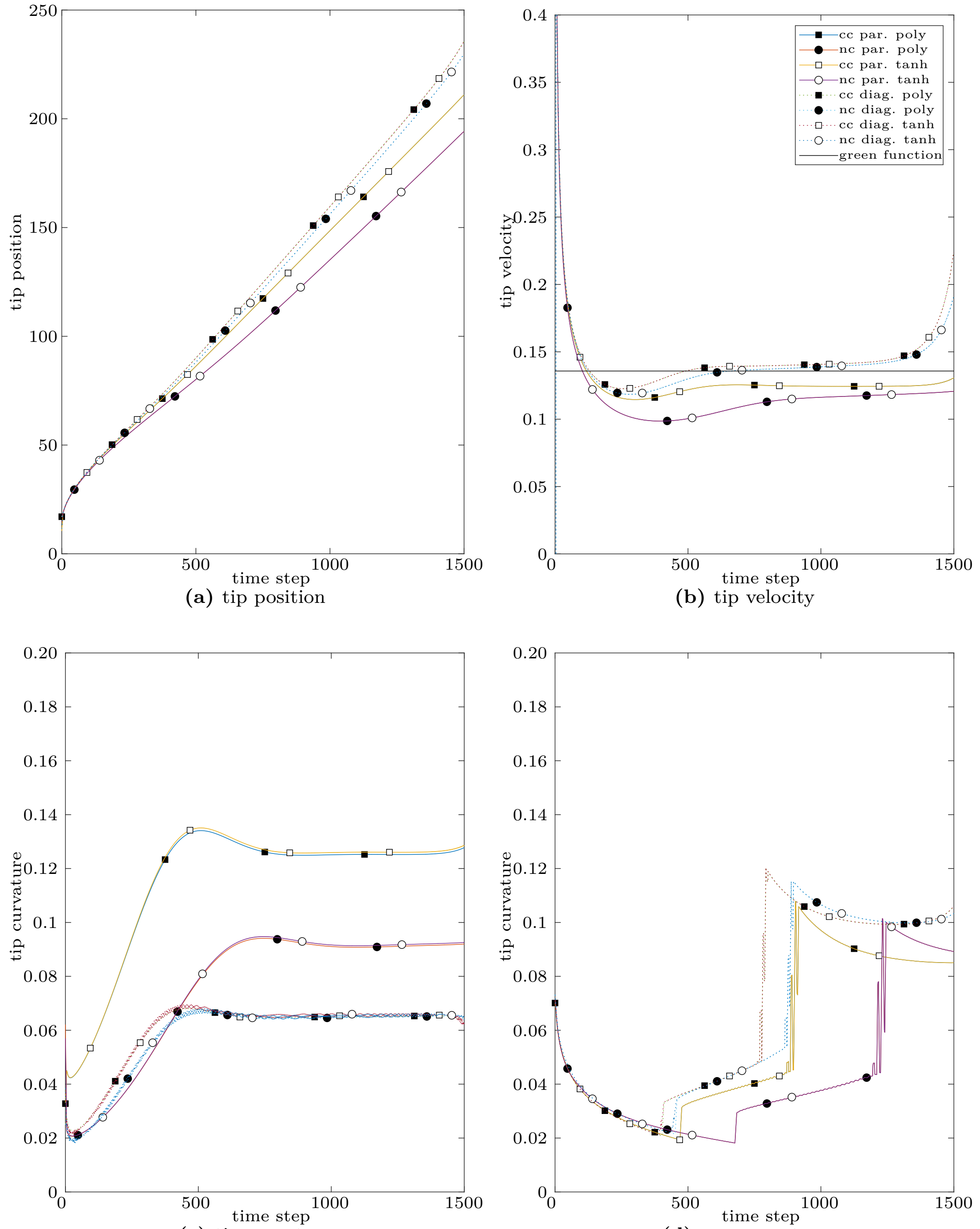


(c) tip curvature



(d) arm curvature

Figure 6.7: Dendirite characterization from the PureMetal simulation using the Problem II set of parameters. Diagonal and parallel growth cases are compared using different post-processing algorithms. Steady-state velocity computed with the green function method is shown for reference in (b).



(a) tip position

(b) tip velocity

(c) tip curvature

(d) arm curvature

Figure 6.8: Dendirite characterization from the PureMetal simulation using the Problem III set of parameters. Diagonal and parallel growth cases are compared using different post-processing algorithms. Steady-state velocity computed with the green function method is shown for reference in (b).

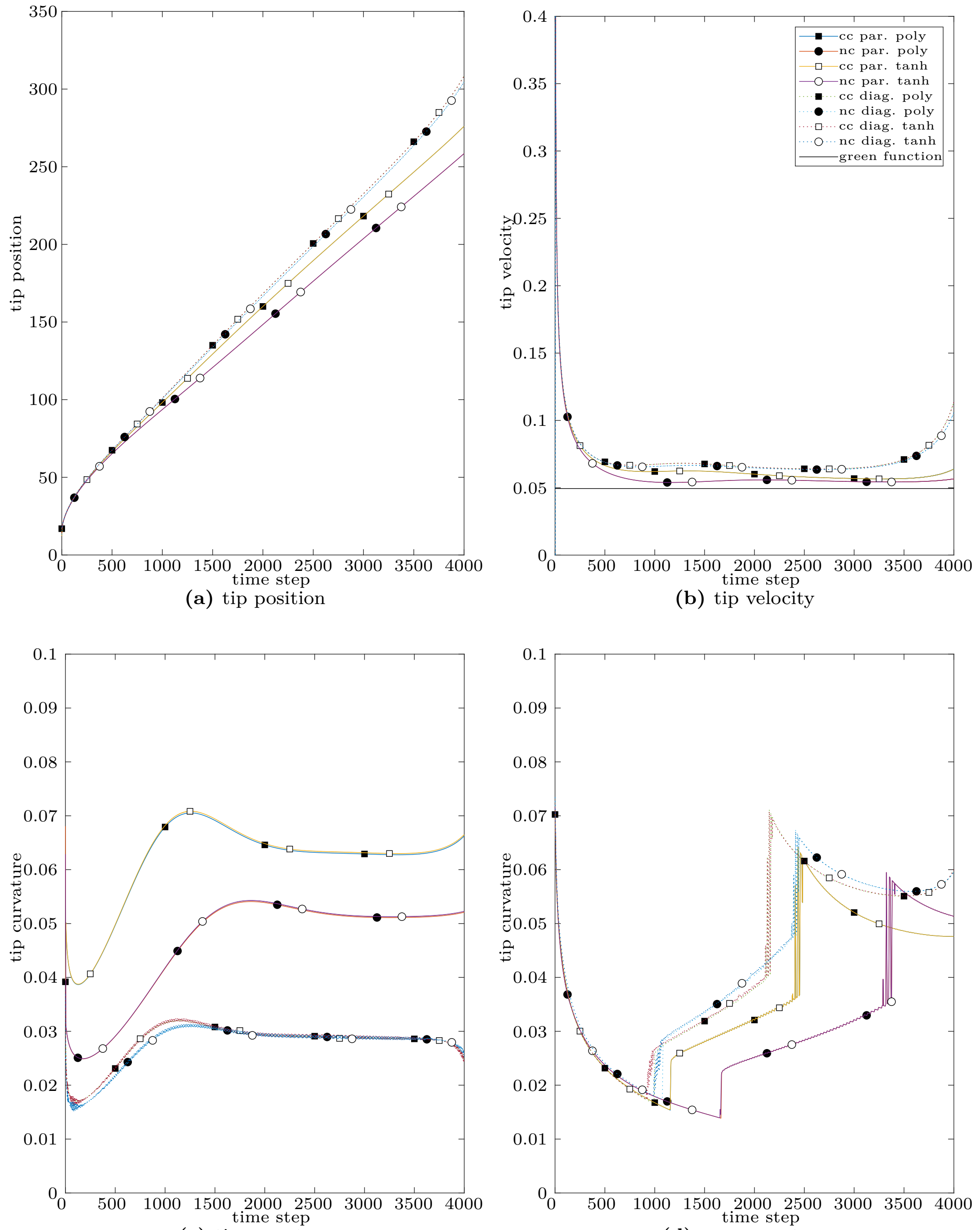


Figure 6.9: Dendirite characterization from the PureMetal simulation using the Problem IV set of parameters. Diagonal and parallel growth cases are compared using different post-processing algorithms. Steady-state velocity computed with the green function method is shown for reference in (b).

	cc diagonal	nc diagonal	cc parallel	nc parallel
I	0.0851 (0.5%)	0.0851 (0.5%)	0.0750 (11%)	0.0750 (11%)
II	0.1272 (4%)	0.1272 (4%)	0.1083 (12%)	0.1083 (12%)
III	0.1372 (1%)	0.1398 (2%)	0.1167 (14%)	0.1244 (8%)
IV	0.06359 (29%)	0.6397 (29%)	0.05434 (10%)	0.05660 (15%)

Table 6.4: Asymptotic tip velocities and their error (in parenthesis) relative to steady-state velocity obtained by Green's function calculations.

provided in [49].

Grid anisotropy, is evident also for the other Problems II–IV and from Figures 6.7–6.9.a can be observed that the dendrite is always growing slower in the parallel than in the diagonal case. Moreover, in the parallel case, cell centered simulations provide less accurate positions and velocity estimates because, in addition to the poor spatial resolution, in this case also no grid location lies on the axis along which the dendrite growth is favored and on which the dendritic arm tip is found.

For Problems II and III, asymptotic tip velocities computed from diagonal simulations are closer to the steady-state values computed using the Green function method when compared with their values computed when $\varepsilon > 0$ while the parallel case predicts velocity in better accordance with Green function computations for Problem IV.

In Table 6.4 a summary of the computed asymptotic velocities for the for problems is given together with the error relative to the steady-state velocity from [49]. Since the choice of the post-processing algorithm does not affect appreciably the value of the tip velocity, values reported therein are distinguished only by their direction and variable basis.

Boundary effects are evident also in Problem II in the diagonal case when after timestep 1700 when the tip position is about 240 spatial units from the origin which is about 70 units from the top and right domain boundaries. Also in Problem III boundary effects start to become evident when the tip is about 75 units from the upper boundaries, both in the parallel and in the diagonal case. For Problem IV, instead, the tip profiles are perturbed by appreciable boundary effects when the tip is less then 90 spatial units away from the upper boundaries.

The above values suggest that when choosing the computational domain size for PureMetal simulations, this has to be large enough to avoid boundary effect and that the margin to be left between position of the tip at the end of the simulation and the upper boundaries is dependent on the problem parameters.

Moreover, in the general case, a diagonal growth produces more accurate result, since it is less dependant on the grid resolution. Smaller values of the anisotropy parameter also require finer grids to avoid larger discretisation errors.

Chapter 7

PhaseField Component AMR Fine/Coarse Interfaces

To allow the use of Adaptive Mesh Refinement (AMR) techniques within applications of the *PhaseField* component, continuity conditions between grid layers with different levels of refinement are to be implemented. All algorithms required for the generation and update of the composite grid resulting from AMR are already provided by the UCF. However, analogously as for boundary conditions, no general implementation is provided for handling artificial boundaries of finer levels. The aim of this step of the development is their implementation within the same general framework used for boundary conditions. An analysis of which features to implement in this step is given in §7.1 while a detailed description of their implementation follows in §7.2.

At last, different approaches will be considered and their performance assessed by comparing the convergence of the solution provided by the `Heat`, `PureMetal` and `Benchmark02` as reported in §7.3.

7.1 Analysis

As already discussed in §4.2.3, three additional steps are to be performed periodically during a simulation in order to use AMR in Uintah's components. These are:

error estimation in this step it is chosen which patches need refinement and which can be coarsened. This choice is controlled by a cell–centered integer valued variable, `RefineFlag`, and a per patch boolean variable, `PatchFlag`. Where the grid resolution could be insufficient to ensure sufficiently accurate solutions, the first variable should assume non-zero values and the second should be set to true.

variable refinement when new patches are added to refined levels, in the data-warehouse no value exists for any of the variables over them. This step is intended to allow the application developer to compute any value that may be necessary for the execution of tasks for time advancing the solution. As such, this step is performed after each geometry update.

variable coarsening this step is introduced to allow the developer to update, if needed, the values of variables at grid cells that have been further refined. In fact, depending on the application, tasks for time advancing the solution may update variables' values only on the most refined regions or may also need to be able to evaluate them on each level. This step is performed at the end of each time step.

The execution order of the steps above is controlled by Unitah's scheduler according to the specifications coded by the application developer in the corresponding scheduling methods of each application.

Via the input file, the final user can control how often the geometry regridding process, and thus the first two steps, may occur as well as other parameters that control how refinement is performed.

Previously, when in §4.2.3 support to AMR has been introduced, it has been necessary to modify most of the methods used for time advancing the solution. Not only did schedulings require new implementations for specifying that on refined patches some task required access to coarser levels, but also tasks needed new ad-hoc implementations for refined patches. These were, first, computing the region on the coarser level where variable values had to be retrieved from the data-warehouse, and, then, they were actually instantiating one or more variables to access and retrieve such values before invoking a kernel. However, most kernels (those involving finite difference computations) had to be modified as well in order to handle fine coarse interfaces.

The main target of this development step is to reduce the work required to the application developer to introduce AMR support in their components by removing, as already done for boundary conditions, the need to implement ad-hoc kernels for handling finite difference approximations at fine coarse interfaces. Differences between the implementation of kernels for inner and physical boundary's grid elements arise, on a refined patch, when the finite difference stencil extends out of the boundaries of a refined level but no boundary condition can be used to extrapolate the missing values. In analogy to conditions on a physical boundary, ghost elements can still be used but the corresponding variable values have to be approximated, instead of imposing a boundary condition, by interpolation of its values at nearby grid elements on the coarser level. This analogy allows the use of the same framework introduced in §6.2. New view type instances can be easily introduced to approximate spatial derivatives at fine coarse interfaces by implementing appropriate `detail::bc_fd` template specializations. These type instances will then need to be explicitly instantiated with automatically generated code as for their boundary condition counterparts. The same factory creational pattern which allows boundary conditions to be chosen by the final user in the input file will provide the user with an analogue method to choose which fine/coarse interface (*FCI*) scheme to use as well.

Moreover, this implementation will result, not only in lightening the burden for application developers and providing better control to final users, but also in a more efficient time advance phase, and in removing any duplication of task methods. The first improvement will be the consequence of AMR views being instantiated only after each geometry update rather that at each time step. The second one will derive from the local nature of the `detail::bc_fd` implementations; in fact, to achieve the same locality characterizing their boundary conditions counterparts, it will be necessary to access information from coarser levels only through views providing an interface that is unaware of any complexity in the coarser level geometry.

Modifying the `detail::get_bc` functor to mark as boundary conditions also fine coarse interfaces will make the `detail::partition_range` functor create a subproblem also for each artificial boundary problem on refined patches. These artificial boundary problems are almost identical to the physical boundary ones, except for their variable views over the refined region which may use `detail::bc_fd` implementations that will contain also instances of views over coarser regions.

By ensuring that refined patches are partitioned after those on the previous (coarser) level, it will be possible to create these coarser view instances by copying the instances already created for the coarser level subproblems, thanks to the prototypical creational pattern provided by views. In this way, refined views can avoid processing again any information about the coarser geometry since coarser views already store all the information required for retrieving and data that may be needed from the data-warehouse. Since data retrieval for coarser views is triggered when their owners `detail::bc_fd` views access the data-warehouse, it will no longer be necessary to explicitly access coarser levels' data from applications tasks on refined patches. It will possible, therefore, to use the same implementation for task methods both on refined and unrefined patches.

It has been found that interpolation is required both in the variable refinement and time advance phases of AMR simulation. It is therefore desirable to provide a common interface to interpolation methods for use both by the framework developer in the `detail::bc_fd` implementation and by the application developer in the refine application tasks. The `AMRInterpolator` template interfaces will be provided as such common interface to interpolate values from coarser levels onto refined positions. Analogously, the `AMRRestrictor` will be introduced for the inverse problem of using refined values to compute the value of variables over coarser grid entries. These template interfaces will allow a simpler implementation of the variable refinement and coarsening steps for the application developer, which provides also a way to isolate the particular implementation of restriction/interpolation operators from their application in simulation components, better addressing the component–based design of the overall UCF.

7.2 Implementation

7.2.1 Interpolators/Restrictors

In the previous analysis it has been shown that isolating the implementation of interpolation and restriction operators is desirable. The `AMRInterpolator` and `AMRRestrictor` template interfaces are provided to the application developer to access such implementations. They are both name aliases of their respective implementations in the `detail` namespace. These aliases are templated, similarly to the `BCFDView` template interfaces, upon the problem type and the index identifying which position in the problem list of boundary variables correspond to the field being interpolated. Instead of the variadic list of boundary packs used to identify the boundary condition, the interpolation order is used as a non-type template parameter. This choice of template parameters allows the application developer to use the `AMRInterpolator` and `AMRRestrictor` template interfaces only with variables in the list of boundary variables of the problem type. This could appear as a restrictive choice since boundary conditions are normally specified for those fields that appear differentiated in the physical model in order to be able to compute their derivatives at physical boundaries, while interpolation may be defined on any variable. Interpolators, as well as restrictors, however use multiple grid values to approximate field values at generic locations and, when these locations are close to a physical boundary, it could happen that some of these values should correspond to a grid position that is actually lying out of the physical domain and are therefore not available. To address this issue, the same interpolation algorithms can be used on any grid entry, treating such grid entries as ghost entries and using the appropriate boundary conditions to extrapolate any missing variable value.

Four distinct specializations for `ScalarFields` are provided for the `detail::amr_interpolator` and two are provided for the `detail::amr_restrictor`, all of which implement the virtual `detail::view` interface for `ScalarFields`. The first of these `amr_interpolator` implementations is independent of the problem dimension and corresponds to the piecewise constant interpolator; its square bracket operator returns, for a given coarse grid index, the value of the underlying variable at the corresponding index on the coarser level. This coarser index is computed by calling the `get_coarser` static member of the `AMRInterface` utility class.

The `AMRInterface` is the analogue of the `DWInterface` utility class and, as such, it is parametrized upon the variable type and problem dimension. It offers two static methods for moving from coarse to refined levels and vice-versa.

To complete the implementation of the square bracket operator for the piecewise constant interpolator, it is necessary to implement an appropriate view for accessing the data-warehouse at coarse indices as computed by the `AMRInterface`. This composite coarse view is implemented in the `detail::amr_coarser_view`.

The `detail::amr_coarser_view`, as for all other `view` implementations for `ScalarFields`, can be created seting the support to be the coarse region underlying a given refined patch and retrieving immediately the corresponding coarse data from the specified data-warehouse, or by seting only the variable and subproblem labels, as well as the material index, which postpones the data retrieval to a later moment when the set method will be used to specify its support.

In the `create_views` body, the `AMRInterface` interface is used to compute the coarse indices identifying the rectangular region underlying the given fine region and, when needed, these coarse indices are modified to be sure that all coarse grid entries that may be required are included. A list of all coarse patches intersecting this coarse region is then retrieved by calling the `selectPatches` method of the coarse level. For each of the coarse patches in this list, the corresponding subproblem partition is retrieved from the new data-warehouse. From each subproblem in this partition whose subregion intersects the requested coarse region, the view instance for the request is accessed, cloned and added to the `detail::piecewise_view` `m_view` list. Immediately after being added at the end of the list, the view clones' supports are also set to the intersection between their subregion and the requested region as the `detail::amr_coarser_view` `m_support` member is modified to include this intersection.

The other three implementations of the interpolator provided, one for each possible problem dimension, define three corresponding partial specializations of the template `detail::amr_interpolator` interface for linear interpolation. As for the piecewise constant implementation, these specializations also use a `detail::amr_coarser_view` instance to access values from the coarser level. The differences with it are in the implementation of the coarse data retrieval and of the square bracket operator.

The square bracket operator for the one-dimensional linear interpolator interpolates the variable values at the two coarser indices whose position are closest to that of the requested fine index. For higher dimensions this process is performed, first, on the edges parallel to the lowest direction to approximate the value of the variable at the orthogonal projection of the requested fine location over those edges. These computed values are then used to linearly interpolate along the next direction until the last direction allowed by the problem dimension is reached. In Figure 7.1 a graphical representation of the piecewise constant and linear interpolators is given for a bidimensional point. The interpolation point, $\boldsymbol{x}_{\bar{\imath},\bar{\jmath}}^{l+1}$ on the refined level $l+1$ is intentionally drawn not at the baricenter of the four nearest grid entries on the coarser level l to stress out that the refinement ratio in each direction is arbitrary and not necessarily equal to 2. In the example the indices on the coarser level l are chosen to have $x_i^l \leq x_{\bar{\imath}}^{l+1} < x_{i+1}^l$ and $y_j^l \leq y_{\bar{\jmath}}^{l+1} < y_{j+1}^l$, in analogy to the indexing criteria used in the `detail::amr_interpolator` implementation of the square bracket operator. In the example, the coarse index closest to the interpolation point, as returned by the `get_coarser` method of the `AMRInterpolator` utility, is $\boldsymbol{x}_{i+1,j+1}^l$, but it may be any of the other three surrounding indices. To retrieve the coordinates relative to both fine and coarse indices, the `get_position` static method of the `DWInterface` is used which, depending on the problem variable basis, returns the coordinate of the centroid or of the lower vertex of the grid cell corresponding to the given index at the given level.

When representing a field discretized with a cell–centered variable, it is common practice to associate each grid value to the whole corresponding cell; the natural choice of interpolation

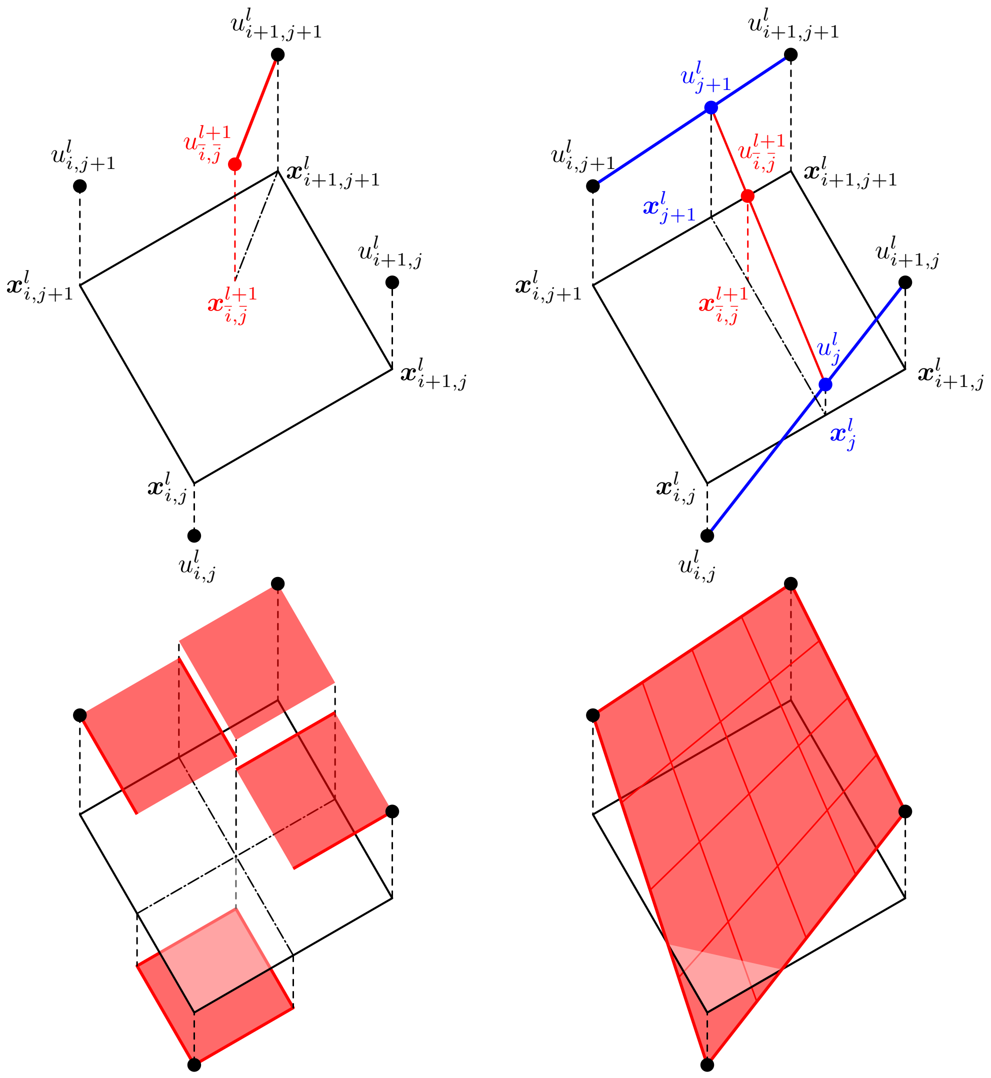

$u^l_{i+1,j+1}$
$u^{l+1}_{\bar{i},\bar{j}}$
$u^l_{i,j+1}$
$\boldsymbol{x}^l_{i+1,j+1}$
$u^l_{i+1,j}$
$\boldsymbol{x}^l_{i,j+1}$
$\boldsymbol{x}^{l+1}_{\bar{i},\bar{j}}$
$\boldsymbol{x}^l_{i+1,j}$
$\boldsymbol{x}^l_{i,j}$
$u^l_{i,j}$
u^l_{j+1}
$\boldsymbol{x}^l_{j+1}$
u^l_j
$\boldsymbol{x}^l_j$

Figure 7.1: Piecewise constant (left) and linear (right) interpolation schemes for a bidimensional problem. On the top, a graphical representation of the two schemes: in this example the interpolated value, $u^{l+1}_{\bar{i},\bar{j}}$, is equal to the coarser value $u^l_{i+1,j+1}$ when using the piecewise constant interpolator. If, otherwise, the linear interpolator is used, the intermediate 1D linear interpolations (blue), u^l_j and u^l_{j+1}, are used to compute the final interpolated value (red). At the bottom, the graph of the two interpolators.

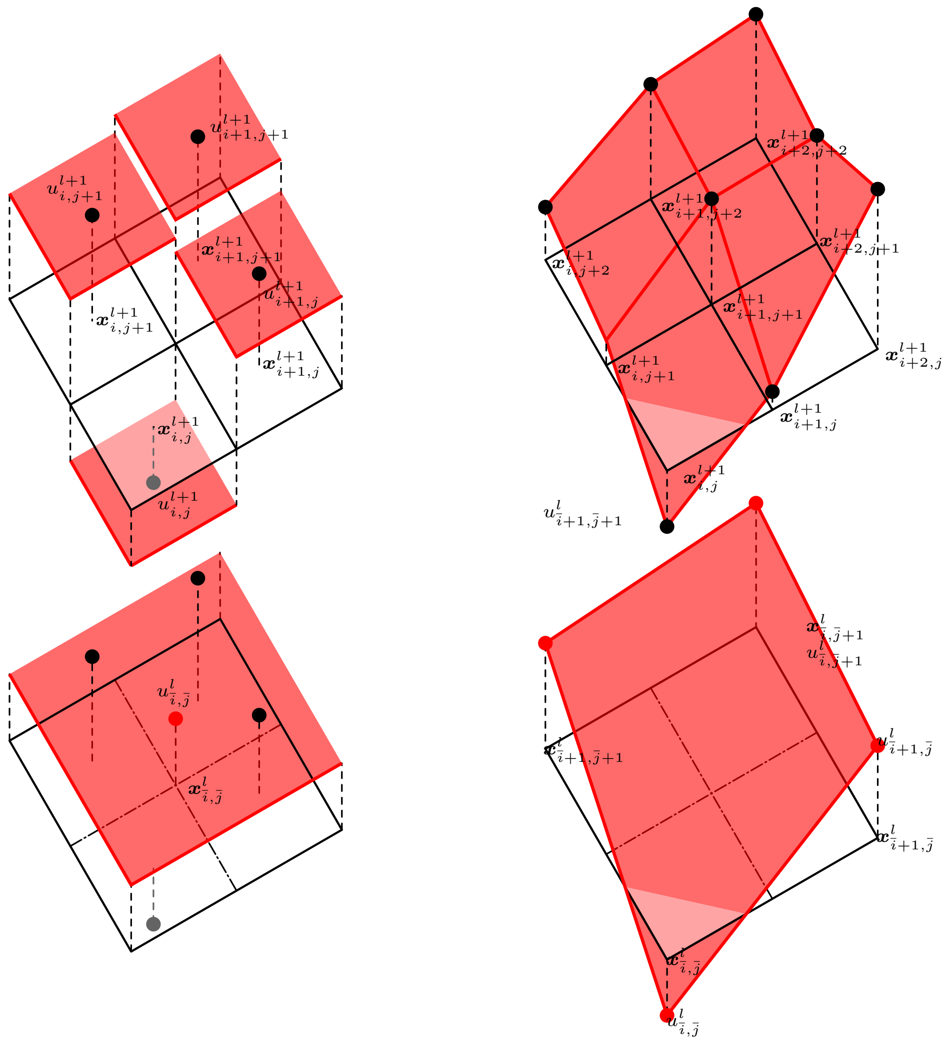

$u^{l+1}_{i+1,j+1}$
$u^{l+1}_{i,j+1}$
$\boldsymbol{x}^{l+1}_{i+1,j+1}$
$u^{l+1}_{i+1,j}$
$\boldsymbol{x}^{l+1}_{i,j+1}$
$\boldsymbol{x}^{l+1}_{i+1,j}$
$\boldsymbol{x}^{l+1}_{i,j}$
$u^{l+1}_{i,j}$
$u^{l}_{\bar{i},\bar{j}}$
$\boldsymbol{x}^{l}_{\bar{i},\bar{j}}$
$\boldsymbol{x}^{l+1}_{i+2,j+2}$
$\boldsymbol{x}^{l+1}_{i+1,j+2}$
$\boldsymbol{x}^{l+1}_{i,j+2}$
$\boldsymbol{x}^{l+1}_{i+2,j+1}$
$\boldsymbol{x}^{l+1}_{i+1,j+1}$
$\boldsymbol{x}^{l+1}_{i+2,j}$
$\boldsymbol{x}^{l+1}_{i,j+1}$
$\boldsymbol{x}^{l+1}_{i+1,j}$
$\boldsymbol{x}^{l+1}_{i,j}$
$u^{l}_{\bar{i}+1,\bar{j}+1}$
$\boldsymbol{x}^{l}_{\bar{i},\bar{j}+1}$
$u^{l}_{\bar{i},\bar{j}+1}$
$\boldsymbol{x}^{l}_{\bar{i}+1,\bar{j}+1}$
$u^{l}_{\bar{i}+1,\bar{j}}$
$\boldsymbol{x}^{l}_{\bar{i}+1,\bar{j}}$
$\boldsymbol{x}^{l}_{\bar{i},\bar{j}}$
$u^{l}_{\bar{i},\bar{j}}$

Figure 7.2: Cell–centered averaging (left) and node centered constant (right) restrictors for a bidimensional problem. The value associated to a coarse cell (left) or to its vertices (right) is computed, in the first case, averaging the values associated to the corresponding refined cells; or, in the second case, using directly the value associated to the coarse vertex at the same location.

scheme in this case is therefore the piecewise constant; otherwise, in the vertex–based case, the linear interpolation is the usual choice.

Similar to `detail::amr_interpolator` are the two implementations provided for the `detail::amr_restrictor` template interface: one, for cell–centered variables, that averages the values at finer levels to compute coarser values and one, for node centered variables, which uses as value at coarse nodes the value at the fine nodes which have the same coordinates.

In Figure 7.2 a representation of the two implementations for two dimensional problems is given. These implementations make use of the `detail::amr_finer_view` template interface which plays the same role played by the `detail::amr_coarser_view` interface in the interpolator implementation. The `detail::amr_finer_view` interface is templated as its counterpart and, as such, its partial specialization for `ScalarFields` implements the `detail::view` interface by publicly inheriting the `detail::piecewise_view` class. Its implementation is symmetrical to the `detail::amr_coarser_view` one.

7.2.2 Fine/Coarse Interfaces schemes

The introduction of artificial boundaries coming with AMR can result in numerical instability. When the finite–difference stencil used to approximate a differential operator at a refined grid element extends out of the refined level grid for a given variable, it is necessary to estimate the values assumed by that variable at those locations that are required to apply the stencil but that are not available in data-warehouse since they do not coincide with the location of any grid entry. As an example, in Figure 7.3 all different possible placements up to rotations of the 5-point stencil over Fine/Coarse Interfaces are shown and entries for which no value is available are drawn in red.

The approximation of these values introduces an additional error to the discretization one. If the estimation strategy is not chosen carefully, this error, which initially is localized at artificial boundaries, can grow and spread to the whole geometry. The natural choice for the estimation strategy is to use the values from coarser levels and adopt the same interpolation scheme that will be used to populate variable values in the data warehouse when new refined patches are created during the simulation. Two interpolation schemes have been introduced for this purpose: a piecewise constant interpolation, which returns the value of the variable at the coarse grid entry closest to the requested fine location; and a bi-linear interpolation using the 2^d closest coarse grid entries.

Figure 7.4 shows the grid entries on the coarser level required by these interpolators for estimating the value of a variable at the refined location pointed by the east entry of the 5-point stencil on the same configurations of the previous example.

As for the previous analysis, for implementing these FCI schemes it has been necessary to implement just a template specialization of the `detail::bc_fd` class template. As stated above, this implementation uses an `AMRInterpolator` instance to compute ghost values. Since the order of interpolation appears as one of the `AMRInterpolator` template parameters, it is necessary to

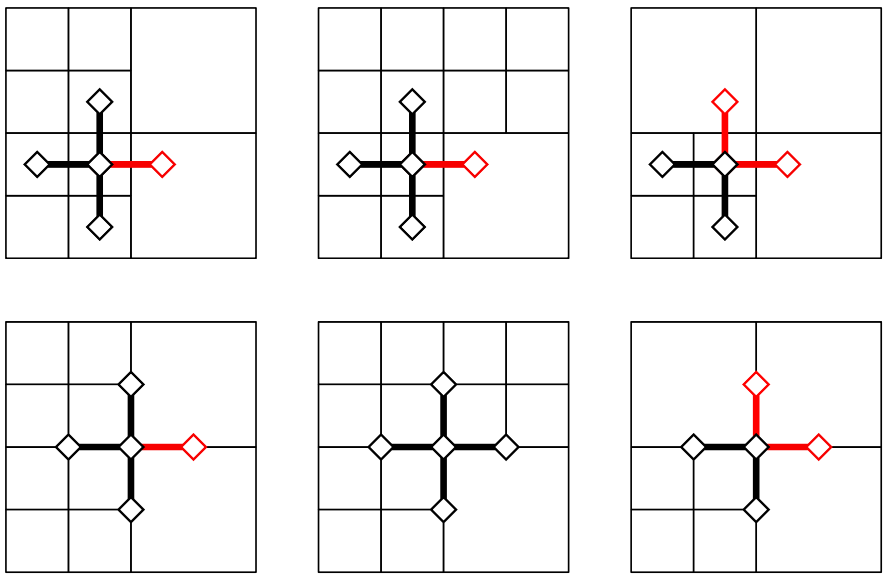


Figure 7.3: Example of different placements of the 5-point stencil over different artificial boundary configurations. Above: cell–centered variable basis case. Below: node centered variable basis case. In red the stencil entries for which variable values need to be estimated.

add to the `detail::bc_fd` template interface an additional template parameter, `C2F`, of type `FC`. Based on it, the value of the interpolation order is deduced at compile

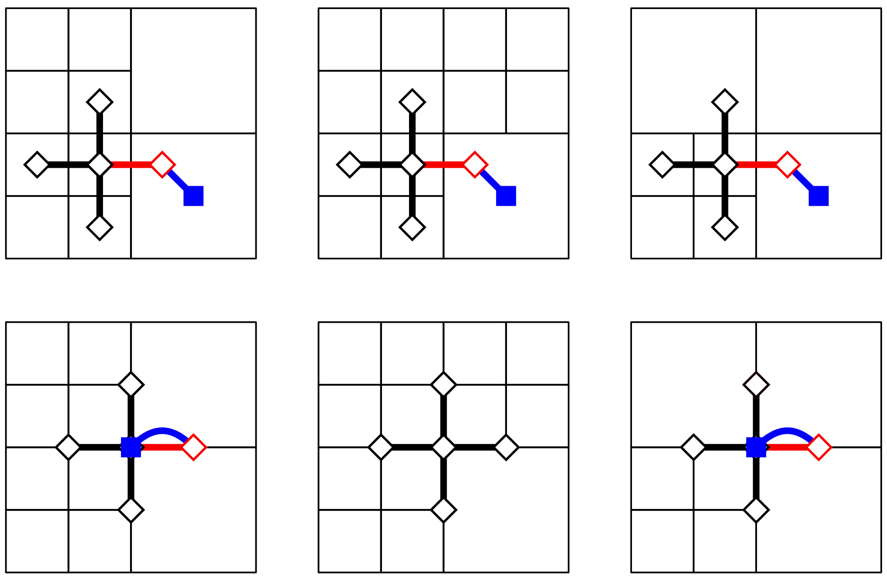
The use of a template interface for the `detail::amr_interpolator` instances allows to define one `detail::bc_fd` partial specialization that provides for both the FCI scheme using piecewise constant interpolation and the linear one.

To allow the use of these schemes, two more steps are necessary. First, the `detail::get_bc` functor implementation has to be modified to handle artificial boundaries; then a script for the generation of the source files containing all explicit instantiations of `BCFDView` type instances has to be provided.

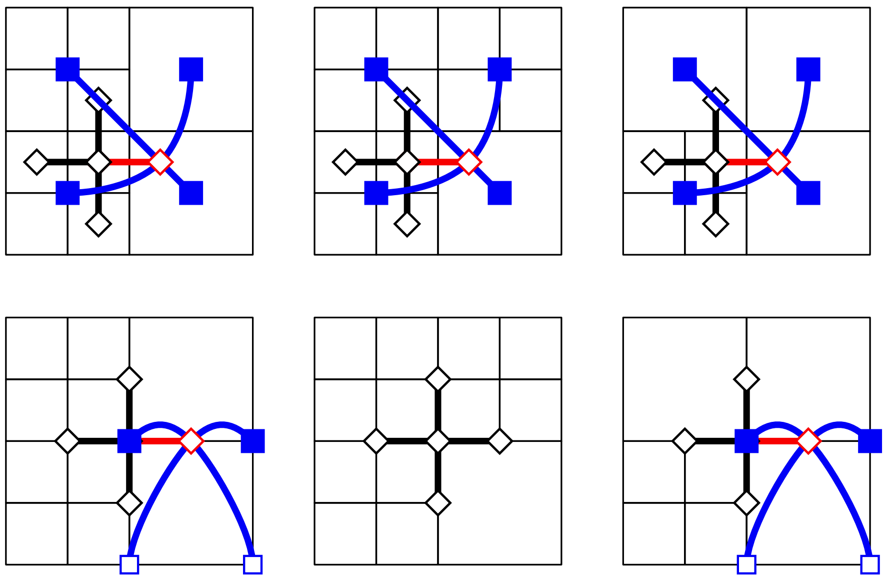
The first step requires to add an additional field, `c2f`, to the `BCInfo` structure that stores all user specification about boundaries as parsed from input files. The parsing of the input file fragment specifying which FCI scheme to use on artificial boundary conditions, has to be implemented by the application developer in the body of the `problemSetup` for each application that supports AMR.

By processing the input file, the `problemSetup` method creates a `std::map` between boundary variable identifiers and FCI schemes to be used by the `detail::get_bc` functor. Its implementation collects together all information about boundary specifications for each given patch and assembles them in a `BCInfo` structure.

The resulting patch boundary information structure can be used by the factory constructor for `BCFDViews` to build at run time the string identifying each boundary configuration and to



(a) Piecewise constant interpolation for cell–centered (above) and node-centered (below) variable basis.



(b) Bi-linear interpolation for cell–centered (above) and node-centered (below) variable basis.

Figure 7.4: Coarser interpolation for 5-point stencil at fine/coarse interfaces. Coarser elements used by interpolators to approximate a variable at the refined location (red diamond) are represented with blue squares. An unfilled square is used for coarser elements with null weights.

instantiate an object of the correct type instance.

All different names identifying the additional `BCFDView` type instances required by AMR simulations, are registered to the `BCFDViewFactory` by their definition in automatically generated source files.

Different estimation schemes to those introduced above, have been implemented by observing that it is possible to use both refined and coarser grid entries rather than only coarse ones as used by interpolators in the estimation of variable values at refined ghost locations. For the sake of simplicity, at this stage only two dimensional alternative schemes for cell–centered variables and refine factors equal to 2 have been investigated; leaving the previous schemes the only available for simulations in three dimensions. In particular, three additional schemes have been identified:

simple 1-D linear interpolation using the values at the two closest grid cells whose centers are aligned with the interpolation point,

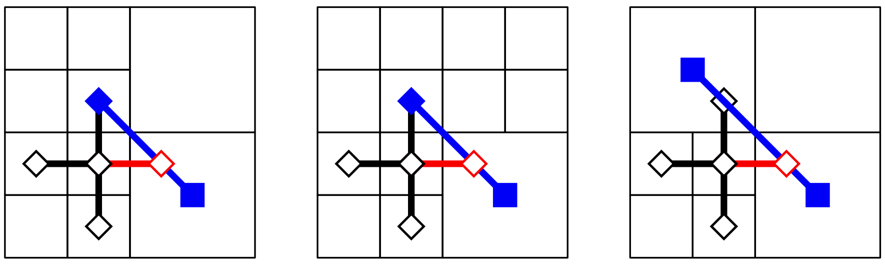
linear 2-D linear interpolation using the values at the three grid cells closest to the interpolation point,

bilinear 2-D bilinear interpolation using the values at the four grid cells closest to the interpolation point.

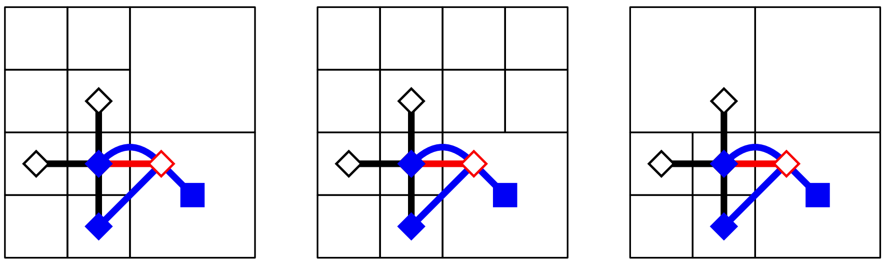
The *simple* FCI scheme can be considered to be an improvement to the previous implementation when using the piecewise constant interpolator. As shown in Figure 7.5(a), the closest available grid entry to the interpolation point in red is still the corresponding coarse cell. To select then the other grid element to use with the 1D interpolation it is possible to follow the direction of the segment joining the coarse cell center and the target point until the next grid element is found. In most configurations, the selected element is one of the fine cells already indicated by the five-point stencil, however, when the finite–difference stencil is centered at the vertex where two artificial boundaries intersect, the selected element is another coarse cell. The fact that two possible cases may apply forces the implementation of the `detail::bc_fd` interface to depend on the surrounding geometry. Since the view used to access the fine level is the same used by all the `detail::basic_fd_view` implementations used by the parent `detail::bcs_basic_fd_view`, the simplest way to check if the first case applies is to check if the fine view support includes the requested fine index or not. In fact, when the selected fine index belongs to a patch different from the one at the center of the stencil, the check using just grid information may not be trivial, while its support would already be set to include the selected fine index, since the fine view will have already retrieved any ghost value from neighbor fine patches before the execution of any `detail::bc_fd` method.

The interpolator instance is created in the same way as it is for the FCI implementation of the `detail::bc_fd` class described earlier.

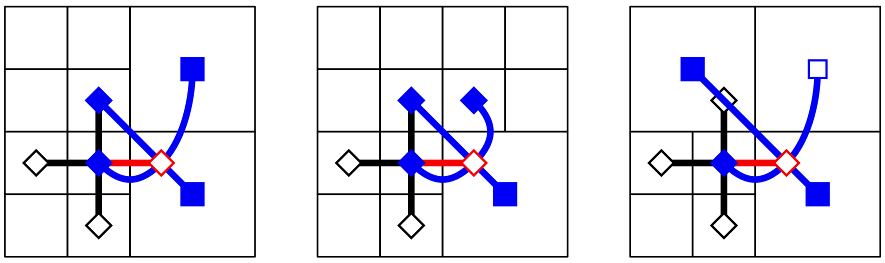
The *linear* FCI scheme is proposed as an improvement to the piecewise constant interpolator scheme with the aim of providing a better estimate of variable values at artificial ghost nodes



(a) Simple FCI scheme.



(b) Linear FCI scheme.



(c) Bi-linear FCI scheme.

Figure 7.5: Approximation schemes for 5-point stencil at fine/coarse interfaces. Coarser elements used by the scheme to approximate a variable at the refined location (red diamond) are represented with blue squares while blue diamonds are used when fine cell values are used. An unfilled square is used for coarser elements with null weights.

without requiring access to any values in addition to those already required by it. Bidimensional linear interpolation is used for this scheme, for which three control points are required. As shown in Figure 7.5(b), the nearest three points to the interpolation one among the center of cells used by the five point-stencil and the piecewise constant interpolator are the coarse one and the stencil center. The stencil point opposite to the target one is the farmost, while the remaining two are equidistant from the interpolation point. Of these two, however, only one is granted to be available in any configuration and is chosen as control point. Choosing the other one, however, would have been equivalent to the simple FCI scheme.

On one hand, since the target point is not included in the convex hull of the control point, this approximation is an extrapolation rather than a proper interpolation, and greater care has to be used when using this scheme; on the other hand, the resulting scheme is independent of the geometry of the fine region which makes this `detail::bc_fd` implementation truly *local*.

At last the *bi-linear* FCI scheme is introduced. The four control points required are chosen to be the closest to the interpolation point whose convex hull includes the requested interpolation point. As shown in Figure 7.5(c), the choice of the control points is dependent on the fine grid geometry and three different cases can be identified.

When the stencil is centered at the intersection of two artificial boundaries, in particular, the coarse control point opposite to the fine region convex vertex can be ignored since its weight is zero.

7.3 Validation

Three different sets of tests have been performed to assess the behavior and the reliability of the FCI schemes implemented in the *PhaseField* component.

The first one aims to pick which of the FCI schemes results in a better control of the error introduced by artificial interfaces. Heat diffusion simulations have been performed on two-levels 2D adaptive grids, so that the error introduced at the artificial interfaces can be compared for the different schemes.

The second set of tests consists in the comparing the output of 2D PureMetal simulations for Problem I (cfr Table 6.2) when different refinement threshold and dilation parameters are used in either cases where the dendritic growth is favored along the carterian axes and along the quadrant diagonals.

7.3.1 Comparison of the implemented FCI schemes

In addition to the two FCI schemes using interpolators to approximate ghost values on a refined grid using values from its coarser level, (piecewise-constant, FC0, and linear, FC1), three additional schemes have been introduced for cell-centered approximations. To asses which FCI scheme introduces the smaller error the artificial interfaces 2D Heat simulations have been performed over 10000 timesteps of size 0.1 on a two-levels adaptive grid over the computational domain $\Omega = [0, 32]^2$ with maximum refinement $h = 1$ using both cell-centered (CC) and vertex-based discretizations (NC). Homogeneous Dirichlet conditions have been imposed on the $x+$ and $y+$ boundaries and homogeneous Neumann conditions on the $x-$ and $y-$ boundaries.

In Table 7.1, the relative error profiles from the different schemes are plotted together with the profiles from simulations on uniform grids with $h = 1$ (labelled *fine*) and $h = 2$ (labeled *coarse*) as references. All relative errors are computed as the ratio between the global errors err_p^N referred to the finest uniform grid and the exact solution norm using (5.7) and (5.11).

Looking at the L^2-relative error profiles, it can be observed how, in the uniform fine cases, this error increase vanishes over time as the solution flattens due to diffusion. The profiles for the uniform coarse cases are instead almost constant over time, since these profiles account mainly for the interpolation error. Since this interpolation is piecewise-constant for cell-centered variables and bi-linear for vertex-based variables, the *CC coarse* profile is higher than the *NC coarse* one.

Focusing on cell-centered profiles, it can be observed that all FCI schemes produces profiles between the two reference uniform profiles, meaning that the error introduced at the artificial interfaces is bounded by the interpolation error. All FCI schemes but the FC0 result in almost identical L^2- and H^1_0-relative error profiles. These are constant over time and about eight times smaller than the reference coarse one. The FC0 L^2 profile, is instead increasing over time: at the beginning of the simulation the L^2-relative error is comparable to the other FCI scheme, but at the end of the simulation it reached the interpolation error. This behavior can be explained looking at the H^1_0 profiles, which provide a measure of the error in the solution derivatives over time. While all other schemes' H^1_0-profiles are bounded by the *CC coarse* one, the *CC FC0* profile is always higher than the reference coarse one and it is also increasing over time. Regridding is affecting appreciably only the H^2_0 profiles: when a regird occurs the interpolation/reduction of the solution over the new grid result in a peak in the profile above the reference *CC FC0* which is rapidly smoothed by diffusion and, overall, all profiles but the *CC FC0* are lower than the *CC coarse one.*

Moving our focus to the vertex-based profiles, these seems to be not bounded by the reference *NC coarse* one. Two factors can be accounted for this undesired behavior: the regularity of the solution, and some issues in how Uintah handles adaptive vertex-based grids. Being the exact solution to the considered problem harmonic, the interpolation error represented by the *NC coarse* profile is much lower than for a generic problem since the interpolation error of the bi-linear interpolator is controlled by the second order derivatives of the solution which is in turn the solution itself times a constant. This is evident by looking at Figure 7.10.a; the local error (5.8) after 500 timesteps has the same spatial distribution as the solution. The analogue plot for the cell-centered coarse reference case Figure 7.9.a shows instead, how the local error, in this case, has the same spatial distribution of the solution's gradient norm.

The second factor is evident in Figure 7.7.a; the artificial interface after 500 iteration is

	NC				CC						
time	coarse	FC0	FC1	fine	coarse	FC0	FCSimple	FCLinear	FCBilinear	FC1	fine
0	$1.51\ 10^{-4}$	$8.54\ 10^{-5}$	$1.02\ 10^{-4}$	$2.03\ 10^{-10}$	$1.11\ 10^{-2}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$2.03\ 10^{-10}$
10	$1.51\ 10^{-4}$	$7.40\ 10^{-3}$	$1.58\ 10^{-4}$	$2.03\ 10^{-8}$	$1.11\ 10^{-2}$	$3.81\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$2.03\ 10^{-8}$
20	$1.51\ 10^{-4}$	$1.34\ 10^{-2}$	$2.16\ 10^{-4}$	$4.06\ 10^{-8}$	$1.11\ 10^{-2}$	$4.52\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$3.79\ 10^{-3}$	$4.06\ 10^{-8}$
30	$1.50\ 10^{-4}$	$1.96\ 10^{-2}$	$2.69\ 10^{-4}$	$6.09\ 10^{-8}$	$1.11\ 10^{-2}$	$5.30\ 10^{-3}$	$3.87\ 10^{-3}$	$3.87\ 10^{-3}$	$3.87\ 10^{-3}$	$3.87\ 10^{-3}$	$6.09\ 10^{-8}$
40	$1.50\ 10^{-4}$	$2.53\ 10^{-2}$	$3.17\ 10^{-4}$	$8.12\ 10^{-8}$	$1.11\ 10^{-2}$	$6.00\ 10^{-3}$	$3.89\ 10^{-3}$	$3.89\ 10^{-3}$	$3.89\ 10^{-3}$	$3.89\ 10^{-3}$	$8.12\ 10^{-8}$
50	$1.50\ 10^{-4}$	$3.10\ 10^{-2}$	$3.64\ 10^{-4}$	$1.01\ 10^{-7}$	$1.11\ 10^{-2}$	$6.80\ 10^{-3}$	$4.02\ 10^{-3}$	$4.02\ 10^{-3}$	$4.02\ 10^{-3}$	$4.02\ 10^{-3}$	$1.01\ 10^{-7}$
60	$1.50\ 10^{-4}$	$3.62\ 10^{-2}$	$4.08\ 10^{-4}$	$1.22\ 10^{-7}$	$1.11\ 10^{-2}$	$7.52\ 10^{-3}$	$4.17\ 10^{-3}$	$4.17\ 10^{-3}$	$4.17\ 10^{-3}$	$4.17\ 10^{-3}$	$1.22\ 10^{-7}$
70	$1.49\ 10^{-4}$	$4.06\ 10^{-2}$	$4.48\ 10^{-4}$	$1.42\ 10^{-7}$	$1.11\ 10^{-2}$	$8.28\ 10^{-3}$	$4.28\ 10^{-3}$	$4.28\ 10^{-3}$	$4.28\ 10^{-3}$	$4.28\ 10^{-3}$	$1.42\ 10^{-7}$
80	$1.49\ 10^{-4}$	$4.44\ 10^{-2}$	$4.99\ 10^{-4}$	$1.62\ 10^{-7}$	$1.11\ 10^{-2}$	$9.07\ 10^{-3}$	$4.32\ 10^{-3}$	$4.32\ 10^{-3}$	$4.32\ 10^{-3}$	$4.32\ 10^{-3}$	$1.62\ 10^{-7}$
90	$1.49\ 10^{-4}$	$4.78\ 10^{-2}$	$5.43\ 10^{-4}$	$1.83\ 10^{-7}$	$1.11\ 10^{-2}$	$9.79\ 10^{-3}$	$4.43\ 10^{-3}$	$4.43\ 10^{-3}$	$4.43\ 10^{-3}$	$4.43\ 10^{-3}$	$1.83\ 10^{-7}$
100	$1.49\ 10^{-4}$	$5.05\ 10^{-2}$	$5.86\ 10^{-4}$	$2.03\ 10^{-7}$	$1.11\ 10^{-2}$	$1.06\ 10^{-2}$	$4.51\ 10^{-3}$	$4.51\ 10^{-3}$	$4.51\ 10^{-3}$	$4.51\ 10^{-3}$	$2.03\ 10^{-7}$

(a) L^2-relative error

	NC				CC						
time	coarse	FC0	FC1	fine	coarse	FC0	FCSimple	FCLinear	FCBilinear	FC1	fine
0	$3.01\ 10^{-4}$	$3.97\ 10^{-2}$	$4.67\ 10^{-4}$	$4.11\ 10^{-5}$	$8.57\ 10^{-3}$	$2.82\ 10^{-2}$	$3.71\ 10^{-3}$	$3.70\ 10^{-3}$	$3.70\ 10^{-3}$	$3.73\ 10^{-3}$	$4.11\ 10^{-5}$
10	$3.01\ 10^{-4}$	$6.22\ 10^{-2}$	$1.06\ 10^{-3}$	$4.11\ 10^{-5}$	$8.57\ 10^{-3}$	$2.77\ 10^{-2}$	$3.84\ 10^{-3}$	$3.75\ 10^{-3}$	$3.81\ 10^{-3}$	$3.91\ 10^{-3}$	$4.11\ 10^{-5}$
20	$3.00\ 10^{-4}$	$8.50\ 10^{-2}$	$1.29\ 10^{-3}$	$4.12\ 10^{-5}$	$8.57\ 10^{-3}$	$3.18\ 10^{-2}$	$3.90\ 10^{-3}$	$3.79\ 10^{-3}$	$3.88\ 10^{-3}$	$4.03\ 10^{-3}$	$4.12\ 10^{-5}$
30	$3.00\ 10^{-4}$	$1.07\ 10^{-1}$	$1.46\ 10^{-3}$	$4.12\ 10^{-5}$	$8.57\ 10^{-3}$	$3.40\ 10^{-2}$	$3.99\ 10^{-3}$	$3.82\ 10^{-3}$	$3.96\ 10^{-3}$	$4.11\ 10^{-3}$	$4.12\ 10^{-5}$
40	$3.00\ 10^{-4}$	$1.38\ 10^{-1}$	$1.57\ 10^{-3}$	$4.12\ 10^{-5}$	$8.57\ 10^{-3}$	$3.51\ 10^{-2}$	$4.03\ 10^{-3}$	$3.84\ 10^{-3}$	$4.01\ 10^{-3}$	$4.22\ 10^{-3}$	$4.12\ 10^{-5}$
50	$2.99\ 10^{-4}$	$1.49\ 10^{-1}$	$1.68\ 10^{-3}$	$4.12\ 10^{-5}$	$8.57\ 10^{-3}$	$3.71\ 10^{-2}$	$4.15\ 10^{-3}$	$3.94\ 10^{-3}$	$4.13\ 10^{-3}$	$4.33\ 10^{-3}$	$4.12\ 10^{-5}$
60	$2.99\ 10^{-4}$	$1.52\ 10^{-1}$	$1.77\ 10^{-3}$	$4.12\ 10^{-5}$	$8.57\ 10^{-3}$	$3.87\ 10^{-2}$	$4.26\ 10^{-3}$	$4.02\ 10^{-3}$	$4.24\ 10^{-3}$	$4.48\ 10^{-3}$	$4.12\ 10^{-5}$
70	$2.99\ 10^{-4}$	$1.62\ 10^{-1}$	$1.83\ 10^{-3}$	$4.13\ 10^{-5}$	$8.57\ 10^{-3}$	$4.01\ 10^{-2}$	$4.33\ 10^{-3}$	$4.07\ 10^{-3}$	$4.31\ 10^{-3}$	$4.55\ 10^{-3}$	$4.13\ 10^{-5}$
80	$2.99\ 10^{-4}$	$1.73\ 10^{-1}$	$1.97\ 10^{-3}$	$4.13\ 10^{-5}$	$8.57\ 10^{-3}$	$4.19\ 10^{-2}$	$4.37\ 10^{-3}$	$4.09\ 10^{-3}$	$4.36\ 10^{-3}$	$4.60\ 10^{-3}$	$4.13\ 10^{-5}$
90	$2.98\ 10^{-4}$	$1.84\ 10^{-1}$	$2.07\ 10^{-3}$	$4.13\ 10^{-5}$	$8.57\ 10^{-3}$	$4.31\ 10^{-2}$	$4.44\ 10^{-3}$	$4.12\ 10^{-3}$	$4.42\ 10^{-3}$	$4.68\ 10^{-3}$	$4.13\ 10^{-5}$
100	$2.98\ 10^{-4}$	$1.95\ 10^{-1}$	$2.15\ 10^{-3}$	$4.13\ 10^{-5}$	$8.57\ 10^{-3}$	$4.52\ 10^{-2}$	$4.47\ 10^{-3}$	$4.16\ 10^{-3}$	$4.46\ 10^{-3}$	$4.72\ 10^{-3}$	$4.13\ 10^{-5}$

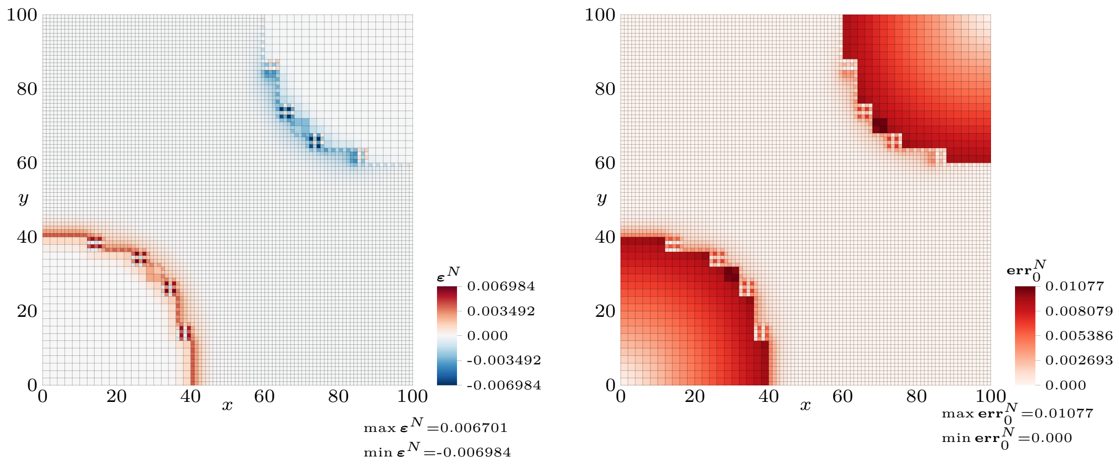
(b) H^1_0-relative error

	NC				CC						
time	coarse	FC0	FC1	fine	coarse	FC0	FCSimple	FCLinear	FCBilinear	FC1	fine
0	$2.22\ 10^{-4}$	$9.42\ 10^{-1}$	$3.34\ 10^{-2}$	$2.06\ 10^{-5}$	$1.32\ 10^{-2}$	$9.24\ 10^{-1}$	$7.35\ 10^{-2}$	$4.90\ 10^{-2}$	$4.61\ 10^{-2}$	$6.84\ 10^{-2}$	$2.06\ 10^{-5}$
10	$2.22\ 10^{-4}$	$9.32\ 10^{-1}$	$1.75\ 10^{-2}$	$2.06\ 10^{-5}$	$1.32\ 10^{-2}$	$2.79\ 10^{-1}$	$9.32\ 10^{-3}$	$4.80\ 10^{-3}$	$9.53\ 10^{-3}$	$1.24\ 10^{-2}$	$2.06\ 10^{-5}$
20	$2.22\ 10^{-4}$	$9.40\ 10^{-1}$	$1.69\ 10^{-2}$	$2.06\ 10^{-5}$	$1.32\ 10^{-2}$	$8.92\ 10^{-1}$	$7.71\ 10^{-3}$	$4.17\ 10^{-3}$	$7.92\ 10^{-3}$	$1.11\ 10^{-2}$	$2.06\ 10^{-5}$
30	$2.21\ 10^{-4}$	$9.48\ 10^{-1}$	$1.70\ 10^{-2}$	$2.06\ 10^{-5}$	$1.32\ 10^{-2}$	$9.65\ 10^{-1}$	$7.38\ 10^{-3}$	$4.13\ 10^{-3}$	$7.52\ 10^{-3}$	$9.40\ 10^{-3}$	$2.06\ 10^{-5}$
40	$2.21\ 10^{-4}$	$9.62\ 10^{-1}$	$1.71\ 10^{-2}$	$2.06\ 10^{-5}$	$1.32\ 10^{-2}$	$1.82\ 10^{-1}$	$7.06\ 10^{-3}$	$4.09\ 10^{-3}$	$7.17\ 10^{-3}$	$1.10\ 10^{-2}$	$2.06\ 10^{-5}$
50	$2.21\ 10^{-4}$	$9.56\ 10^{-1}$	$1.68\ 10^{-2}$	$2.07\ 10^{-5}$	$1.32\ 10^{-2}$	$1.52\ 10^{-1}$	$8.47\ 10^{-3}$	$4.75\ 10^{-3}$	$8.57\ 10^{-3}$	$9.82\ 10^{-3}$	$2.07\ 10^{-5}$
60	$2.21\ 10^{-4}$	$9.60\ 10^{-1}$	$1.68\ 10^{-2}$	$2.07\ 10^{-5}$	$1.32\ 10^{-2}$	$1.57\ 10^{-1}$	$7.88\ 10^{-3}$	$5.18\ 10^{-3}$	$8.03\ 10^{-3}$	$1.14\ 10^{-2}$	$2.07\ 10^{-5}$
70	$2.20\ 10^{-4}$	$9.53\ 10^{-1}$	$1.66\ 10^{-2}$	$2.07\ 10^{-5}$	$1.32\ 10^{-2}$	$1.54\ 10^{-1}$	$7.53\ 10^{-3}$	$4.86\ 10^{-3}$	$7.64\ 10^{-3}$	$9.33\ 10^{-3}$	$2.07\ 10^{-5}$
80	$2.20\ 10^{-4}$	$9.47\ 10^{-1}$	$1.83\ 10^{-2}$	$2.07\ 10^{-5}$	$1.32\ 10^{-2}$	$7.65\ 10^{-1}$	$7.21\ 10^{-3}$	$4.82\ 10^{-3}$	$7.36\ 10^{-3}$	$9.10\ 10^{-3}$	$2.07\ 10^{-5}$
90	$2.20\ 10^{-4}$	$9.58\ 10^{-1}$	$1.77\ 10^{-2}$	$2.07\ 10^{-5}$	$1.32\ 10^{-2}$	$1.62\ 10^{-1}$	$9.08\ 10^{-3}$	$4.23\ 10^{-3}$	$9.20\ 10^{-3}$	$1.08\ 10^{-2}$	$2.07\ 10^{-5}$
100	$2.20\ 10^{-4}$	$9.33\ 10^{-1}$	$1.77\ 10^{-2}$	$2.08\ 10^{-5}$	$1.32\ 10^{-2}$	$6.86\ 10^{-1}$	$7.45\ 10^{-3}$	$4.87\ 10^{-3}$	$7.51\ 10^{-3}$	$8.89\ 10^{-3}$	$2.08\ 10^{-5}$

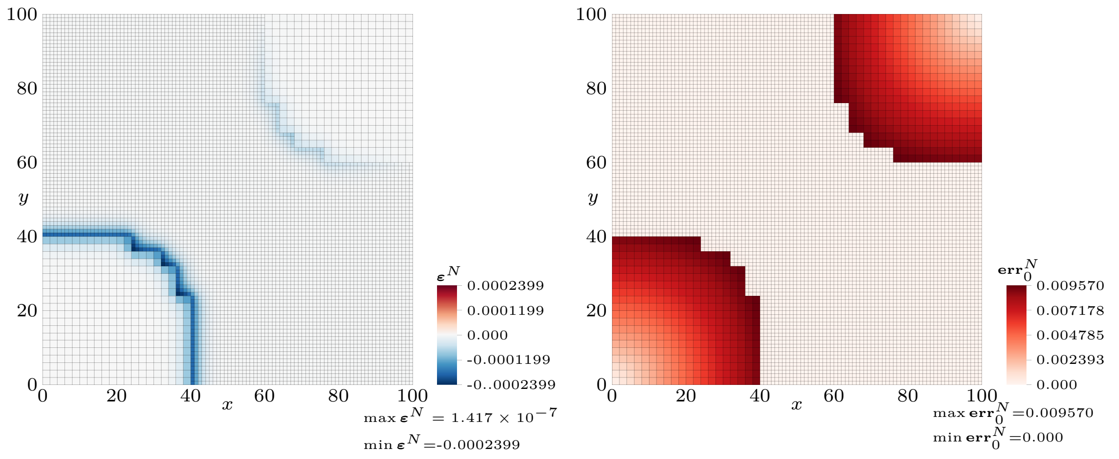
(c) H^2_0-relative error

Table 7.1: Relative errors from Heat simulations for the different FCI schemes.

not symmetrical with respect to the bisector of the first quadrant when the exact solution is. Debugging made possible to find that the issue is related to Uintah not handling correctly ghosts values of the flag variable used to keep track of where refinement is necessary when vertex-based grids are used, which already add additional grid entries to handle nodes on the right/top/above edges.



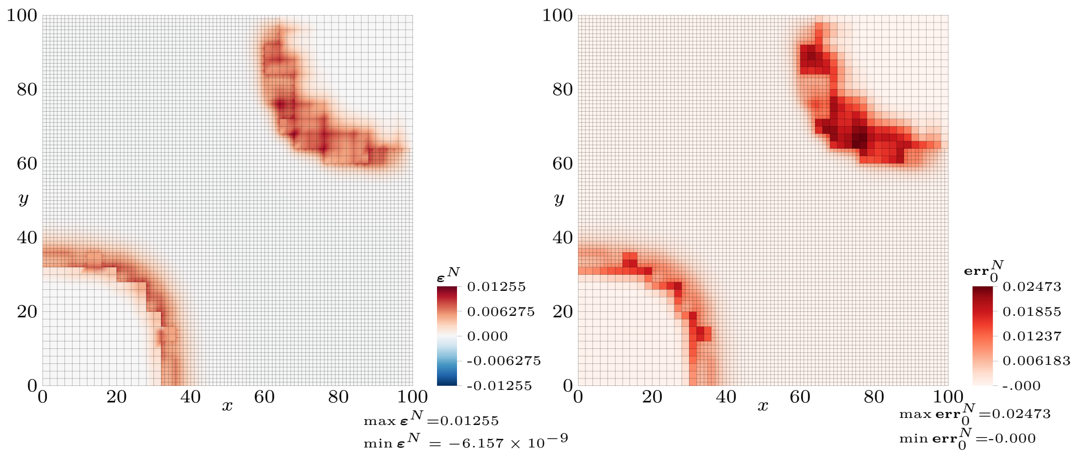
(a) CC FC0



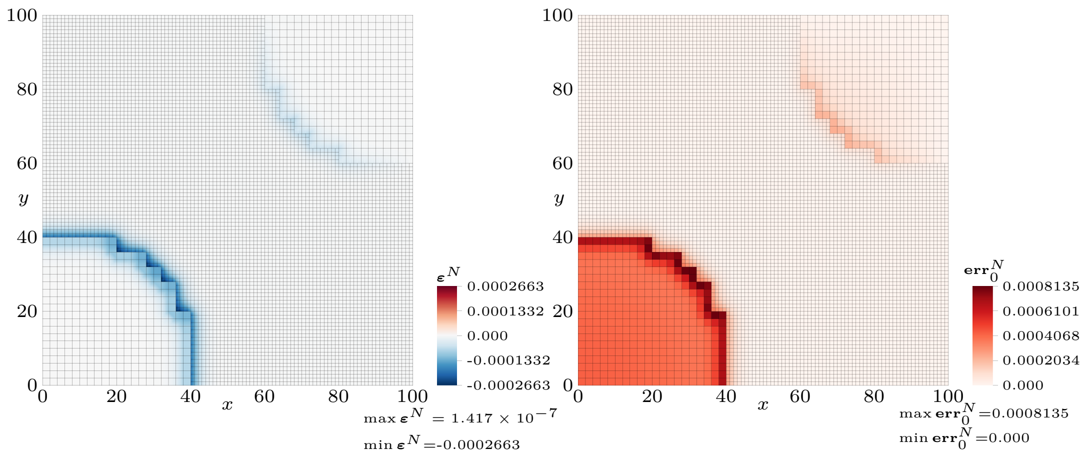
(b) CC FC1

Figure 7.6: Pseudocolor plots of the 0-residual, ε_0^N, (left) and of the local L^2-error, $\mathbf{err}_0^N$ (right) after five timesteps, $t^N = 5$ for Heat simulations using cell-centered variable representation and standard interpolators at fine–coarse interfaces: piecewise-constant (a), and piecewise-linear (b).

For each choice of FCI scheme and variable basis, pseudocolor plots of the residual (5.6) and local L^2-error (5.8) are available in Figures 7.6-7.10. In addition to the previous observations, it can be seen that the residual at the artificial interfaces is not smooth in the cell-centered case when using the piecewise-constant coarse interpolation (FC0), the simple (FCSimple) or the

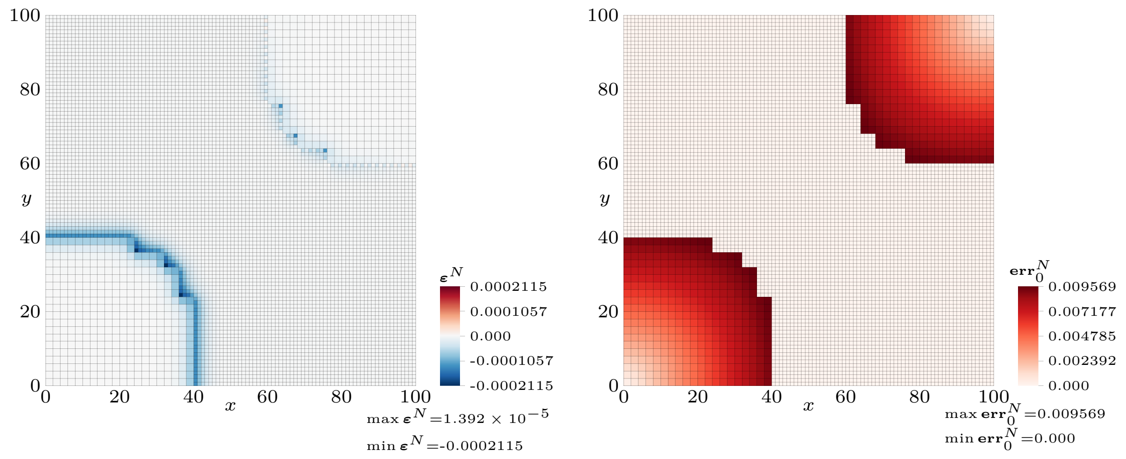


(a) NC FCO

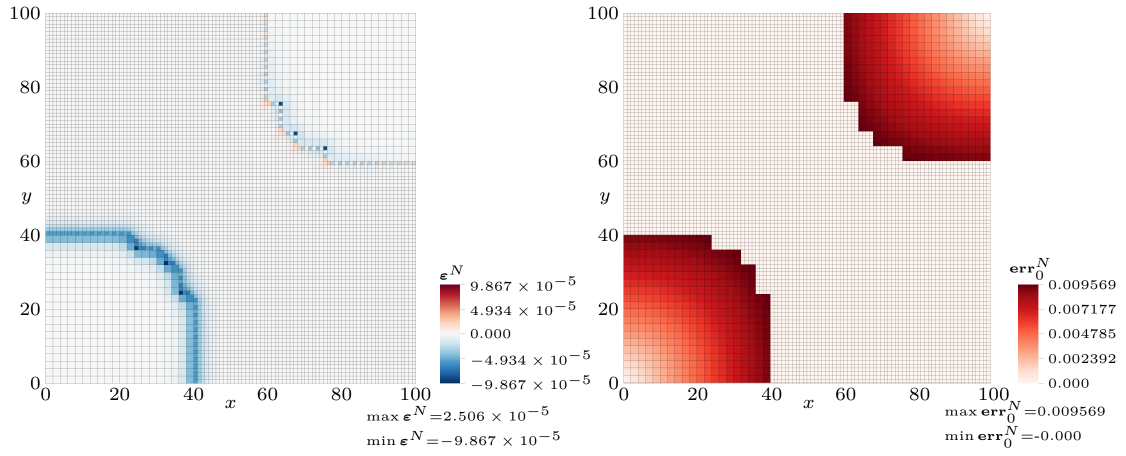


(b) NC FC1

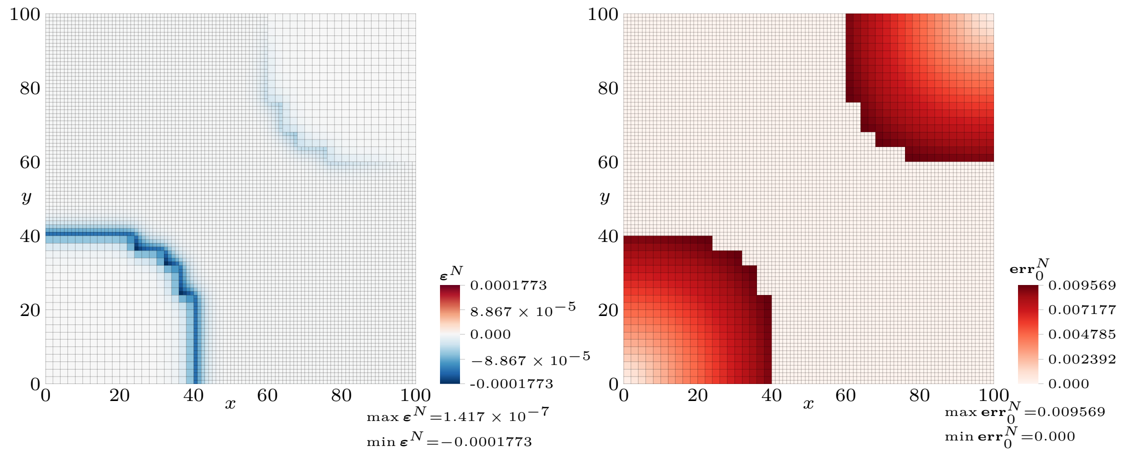
Figure 7.7: Pseudocolor plots of the 0-residual, $\boldsymbol{\varepsilon}_0^N$, (left) and of the local L^2-error, $\mathbf{err}_0^N$ (right) after five timesteps, $t^N = 5$ for Heat simulations using vertex-based variable representation and standard interpolators at fine–coarse interfaces: piecewise-constant (a), and piecewise-linear (b).



(a) CC FCSimple

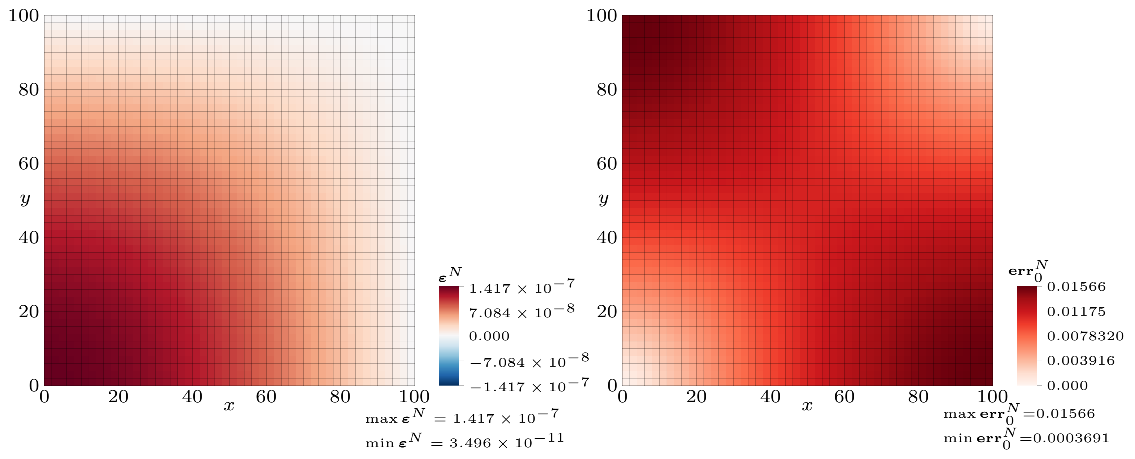


(b) CC FCLinear

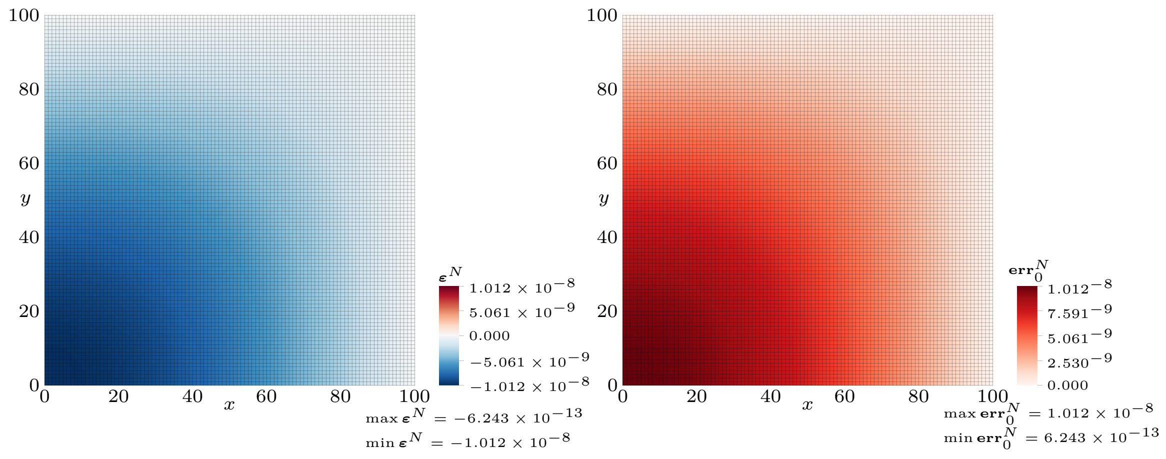


(c) CC FCBilinear

Figure 7.8: Pseudocolor plots of the 0-residual, $\boldsymbol{\varepsilon}_0^N$, (left) and of the local L^2-error, $\mathbf{err}_0^N$ (right) after five timesteps, $t^N = 5$ for Heat simulations using cell-centered variable representation and ad–hoc interpolators at fine–coarse interfaces: simple (a), linear (b) and bilinear (c).

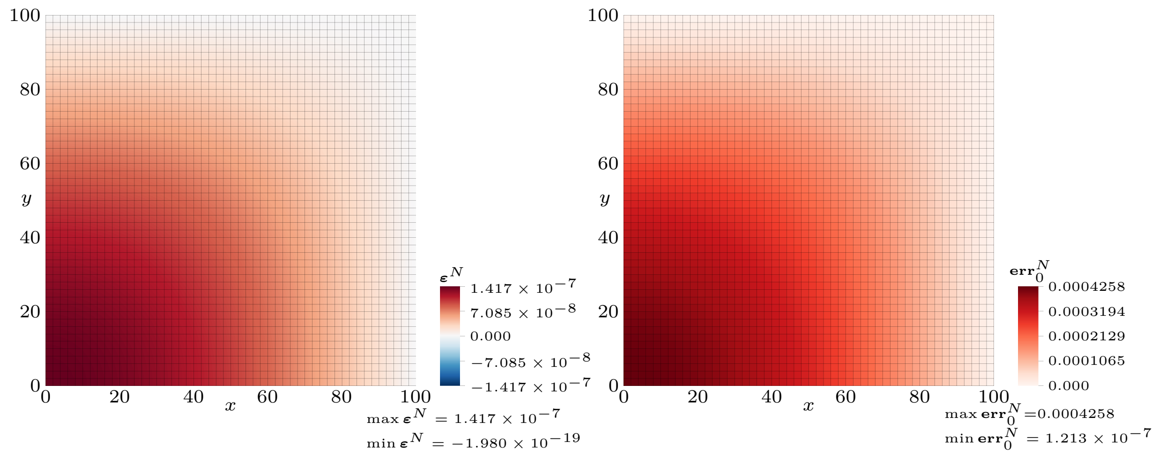


(a) CC uniform coarse

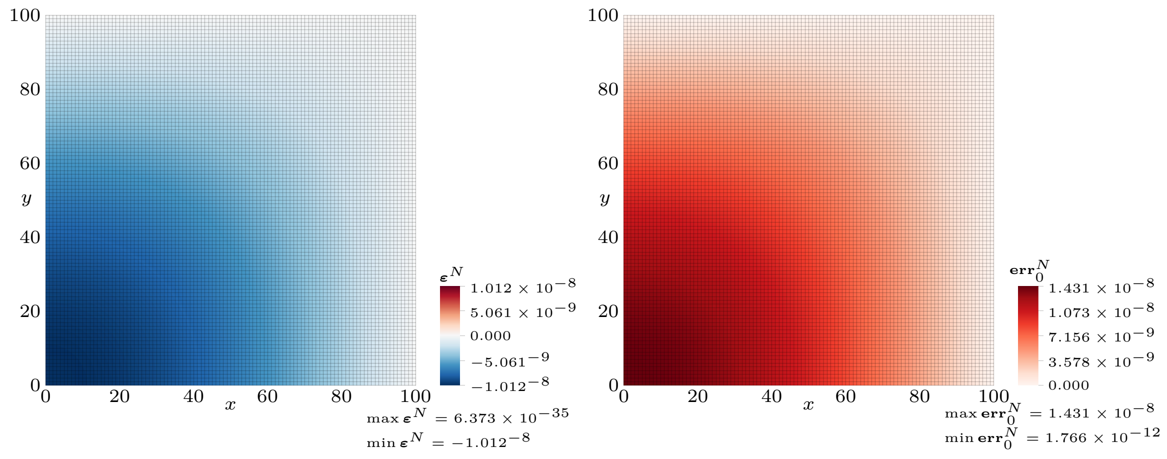


(b) CC uniform fine

Figure 7.9: Pseudocolor plots of the 0-residual, $\boldsymbol{\varepsilon}_0^N$, (left) and of the local L^2-error, $\mathbf{err}_0^N$ (right) after five timesteps, $t^N = 5$ for Heat simulations using cell-centered variable representation over uniform grids: coarse, $h = 2$ (a); and fine, $h = 1$ (b).



(a) NC uniform coarse



(b) NC uniform fine

Figure 7.10: Pseudocolor plots of the 0-residual, $\boldsymbol{\varepsilon}_0^N$, (left) and of the local L^2-error, $\mathbf{err}_0^N$ (right) after five timesteps, $t^N = 5$ for Heat simulations using vertex-based variable representation over uniform grids: coarse, $h = 2$ (a); and fine, $h = 1$ (b).

linear scheme (FCLinear). Moreover the sign of the residual in the FC0 and FCLinear cases has alternating signs on both sides of fine/coarse interfaces.

From the above results, it is evident that when using AMR, only cell-centered discretizations should be used since vertex-based variable representation in conjunction with AMR is still not fully supported by the Uintah Computation Framework. Concerning the choice of the FCI scheme, it can be concluded that the best choices are the FCBilinear and the FC1. Both schemes produce almost identical error profiles for the solution and its first order derivatives, with the latter producing slightly smaller errors in the approximation of the solution's second order derivatives (H_0^2-profiles). For these reasons, in the following tests and AMR simulations cell-centered discretization is used together with the FC1 scheme.

7.3.2 Choice of refinement threshold and dilation parameters

The second set of tests has been conducted to investigate how the solution is affected by the parameters affecting the regridding phase of a simulation. These parameters are the refinement threshold τ which identifies the region, $\mathscr{R}_{\text{finest}}$, of the computational domain that needs to be covered by the finest level of the adaptive grid. This approach does not try to estimate the local error of the solution and choose to refine further the grid where the error estimate is above a given tolerance and coarsen the grid where the error is below that tolerance. The geometry of the levels between the finest and the coarsest are therefore not controlled by τ but only by dilation parameters, that are parameters controlling the number of cells that should exist in a level between the interface with its finer level and the interface with its coarser level. Two parameters control this number: the *stability–dilation*, n_s, and the *regrid–dilation*, n_r, parameters. Both of the two specify the number of cells that should exist in the AMR grid between adjacent artificial interfaces. They differ in the fequency by which the Unitah regridder checks if the grid satisfies these specifications. The first is verified after each timestep, while the latter only after a regrid occurs, as described in §2.2.4. Due to the implementation of FCI schemes described in the previous section which requires each artificial boundary to be between adjacent levels, the stability–dilation parameter must satisfy $n_s \geq 1$.

In the following are reported the results from several 2D PureMetal simulations of Problem I. The first subset of which is intended to investigate the relation between the refine threshold τ and the error in the computation of the asymptotic tip velocity with respect to its value as computed in [49] using the green function method. The same problem has been simulated on AMR grids with maximum number of levels, n_l, in the range[1] 1–8 but keeping fixed the finest grid step. When the favored dendritic growth disection is parallel, the computational domain $\Omega = [0, L]^2$ with $L = 240$ has been used with up to three AMR levels. When more levels are used, the domain has been enlarged to ensure that the domain upper bound satisfied $L = m2^{n_l-1}h_{\text{finest}}$ for some integer m. The same precaution is used in the diagonal case where

[1] 8 is the maximum number of levels allowed by Uintah's regridder

$L = 179.2$ is the minimal upper bound considered. All simulation in this subset used the minimal dilation parameters $n_s = 1$ and $n_r = 0$, while the following refinement threshold have been considered: $\tau = 0.01, 0.005, 0.0025$.

The computed asymptotic velocity values are summarized, together with its errors, in Tables 7.2–7.4. These tables contain also information about the AMR grid for each simulation: the number of regrids occurred, together with the time average, minimum and maximum total number of cells across all levels. Profiles of the tip characteristics and of the total number of cells are also given in Figures 7.11–7.13.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.93106	10.9%	1	360000	360000	360000
	2	0.07323	0.01141	13.4%	2855	100185	90896	108064
	3	0.07021	0.01443	17.0%	2782	36281	23924	46676
	4	0.06671	0.01794	21.1%	2649	20767	7504	31792
	5	0.06343	0.02122	25.0%	2543	16859	3444	27828
	6	0.06215	0.02250	26.5%	2522	16052	2496	26992
	7	0.06200	0.02264	26.7%	2507	15897	2260	26868
	8	0.06200	0.02265	26.7%	2507	15925	2260	26932
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08245	0.00220	2.6%	3046	60837	51104	70128
	3	0.07887	0.00577	6.8%	2969	26788	13968	39168
	4	0.07476	0.00989	11.6%	2846	18534	4880	31712
	5	0.07143	0.01322	15.6%	2761	16572	2816	29728
	6	0.06975	0.01490	17.6%	2693	16181	2308	29268
	7	0.06944	0.01521	17.9%	2712	16181	2240	29392
	8	0.06944	0.01521	17.9%	2712	16197	2256	29408

Table 7.2: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $\tau = 0.01$ for different number of levels, n_l, and minimal dilation parameters, $n_s = 1$ and $n_r = 0$.

Looking at the results obtained for each considered value of τ within each one of Tables 7.2–7.4, it is possible to observe how the value of h_{finest} is not fine enough to avoid grid anisotropy with parallel simulations producing less accurate solution than diagonal ones. However a finer value for h_{finest}, due to the stability constraint on the timestep, would result in unpractical simulation times. It is evident how increasing the number of levels n_l reduces considerabily the total number of cells in the grid up to about a tenth of the size of the uniform grid. For the problem considered, however, the reduction of the overall grid size due to using an additional level decrease sensibly as the n_l increases. It appears that using more than four or five levels is not beneficial in reducing the overall grid size. The number of regrids performed during the simulation seems independent of the number of levels. The number of AMR levels affects instead the error in the tip velocity: as n_l increases a sensible worsening of the tip velocity is observed until a maximum relative error is reached. This behavior may suggest that the finest region $\mathscr{R}_{\text{finest}}$ is to narrow around the phase-field interface. It seem that there exists a value of n_l after which adding a level is coarsening only region that don't affect the quality of the solution. The comparison between Tables 7.2–7.4, however allows to conclude that lowering the refinement threshold does not help to reduce the worsening of the solution as additional AMR levels are introduced. This is due to the way region $\mathscr{R}_{\text{finest}}$ is defined; in fact, the gradient of the phase-field vanishes immediately away from the solid-liquid interface, therefore lowering τ is not enough to widen enough $\mathscr{R}_{\text{finest}}$ around the solid-liquid interface.

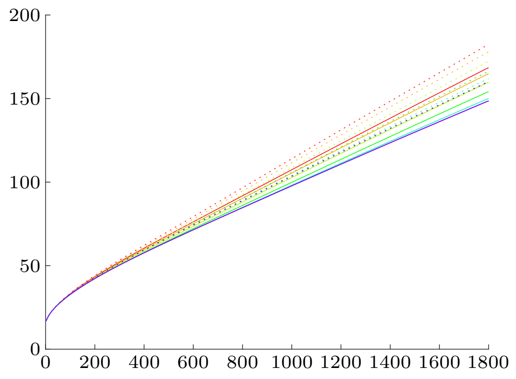
A better approach to improve the quality of the solution when increasing the number of AMR levels would be to increase the width of the finest region around the liquid-solid interface. This can be done by changing the dilation parameters. For this reason the second subset of tests has been conducted to compare the tip velocities computed with different values of n_s. In Uintah there is a hard limit on the overall thickness of the dilation layer when parallel simulations are performed. In the *Core/Parallel/Parallel.h* header file the macro `MAX_HALO_DEPTH` is defined to 5 and is used by the loadbalancer to separate out the standard from the distal ghost cell requirements and the scheduler will then generate two different set of patch neighbors. Uintah's regridder is looking only in the first set of neighbors when dilation is applied (it is assuming that dilation should not involve distal patches to semplify the already expensive regridding step). For this reason parallel simulation where $n_s + n_r <$ `MAX_HALO_DEPTH` may behave unexpectedly when the dilated finest region crosses patches assigned to different processors.

The considered values for n_s are 2 and 4 and the results reported in Tables 7.5 and 7.6 should be compared with Table 7.4 since the refinement tolerance chosen for these simulations is $\tau = 0.0025$ and the refinement dilation parameter is $n_r = 0$. In Figures 7.11–7.13, the profiles of the tip characteristics and of the total number of cells are available. A drawback of this approach is that the same dilation is applied to each level leading to bigger grids than necessary in the current application. Here it is important to dilate only the finest level while on intermediate levels only one cell layer is required by FCI schemes between artificial interfaces.

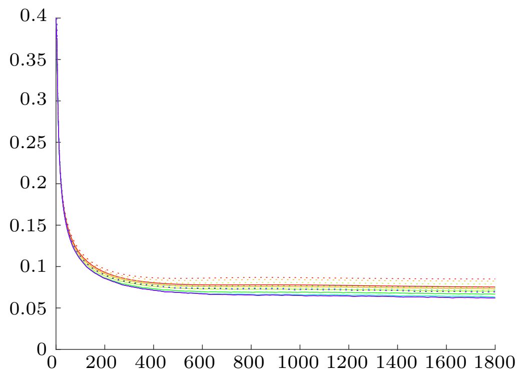
Looking at the total cell maximum values we can observe as incrementing the dilation by one unit result in increasing the AMR grid size by a small factor (3%), but using $n_s = 4$ can increment the grid size almost by half. Also the overall number of regrids is affected by enlarging the dilation layer. Setting $n_s = 4$ can result in the regridding occurring almost 30% more often.

Small benefit in the tip velocity computations is observed for $n_s = 2$ but the error is halved for $n_s = 4$, which confirms that the additional error caused by the introduction of AMR levels observed in the previous subset of test is due to the finest region being to narrow around the solid-liquid interface.

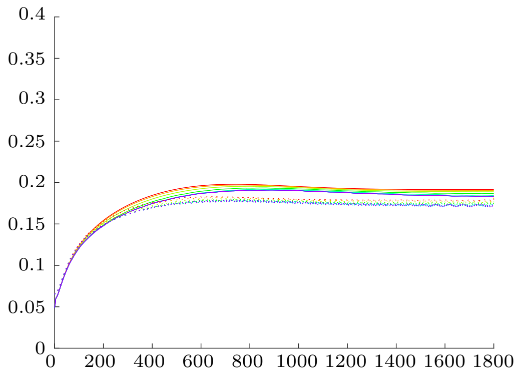
However, since the regridding phase can be quite computationally expensive, it can be useful to check if a smaller error can be achieved by increasing the regrid-dilation parameter instead of n_s. This parameter enlarges the finest region only after a regrid and then, as the simulation progresses and the physical interface moves forward, the layer of cells between the physical and



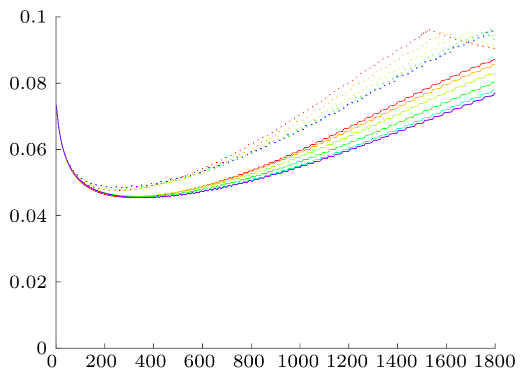
(a) tip position



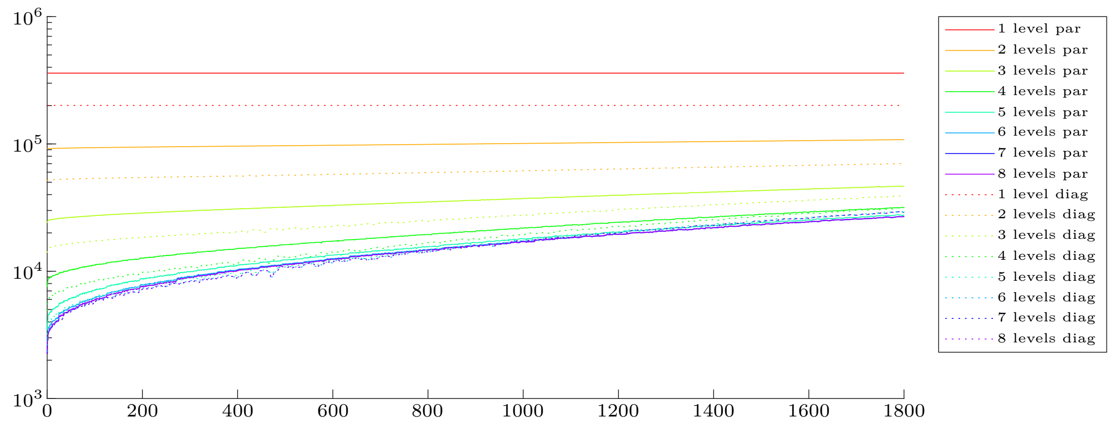
(b) tip velocity



(c) tip curvature

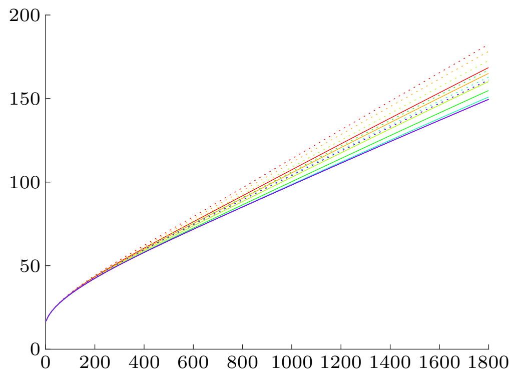


(d) arm curvature

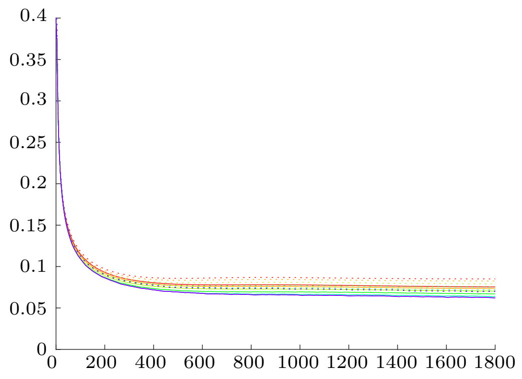


(e) grid cells

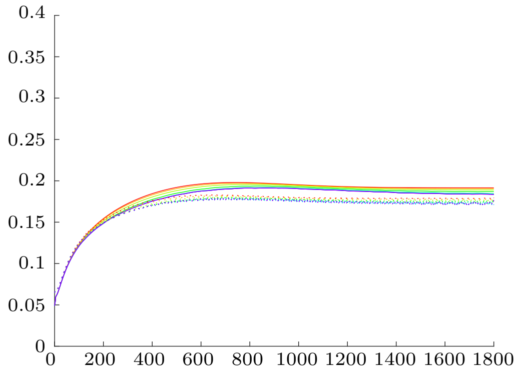
Figure 7.11: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $\tau = 0.01$ for different number of levels, n_l, and minimal dilation parameters, $n_s = 1$ and $n_r = 0$.



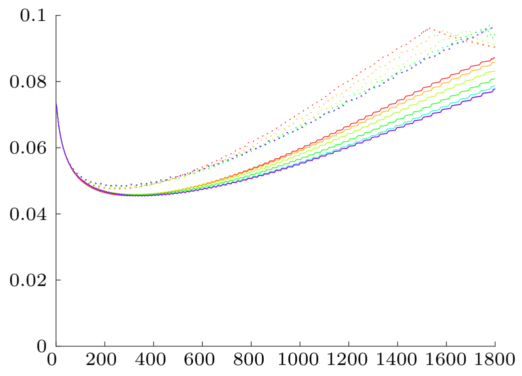
(a) tip position



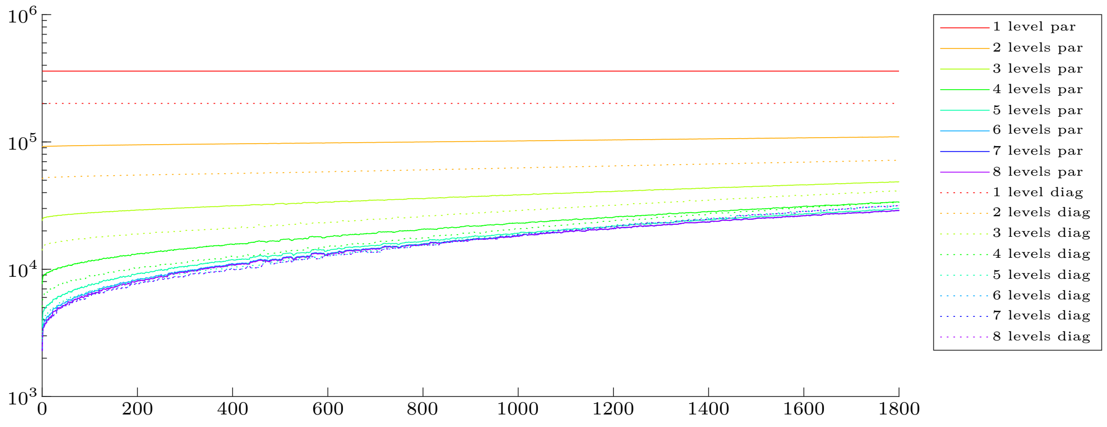
(b) tip velocity



(c) tip curvature

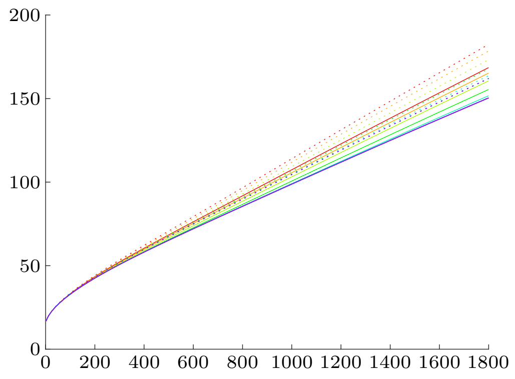


(d) arm curvature

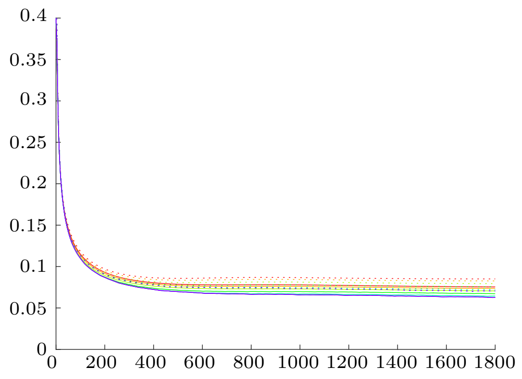


(e) grid cells

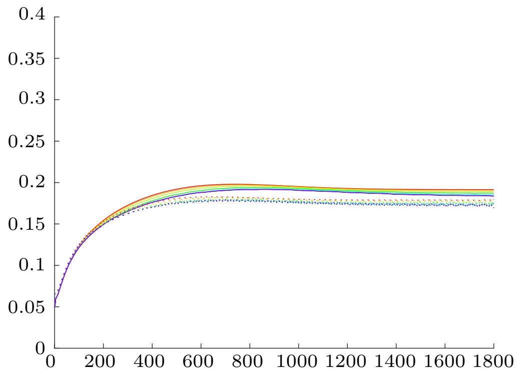
Figure 7.12: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $\tau = 0.01$ for different number of levels, n_l, and minimal dilation parameters, $n_s = 1$ and $n_r = 0$.



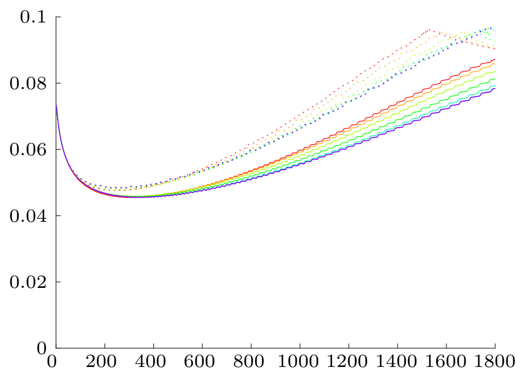
(a) tip position



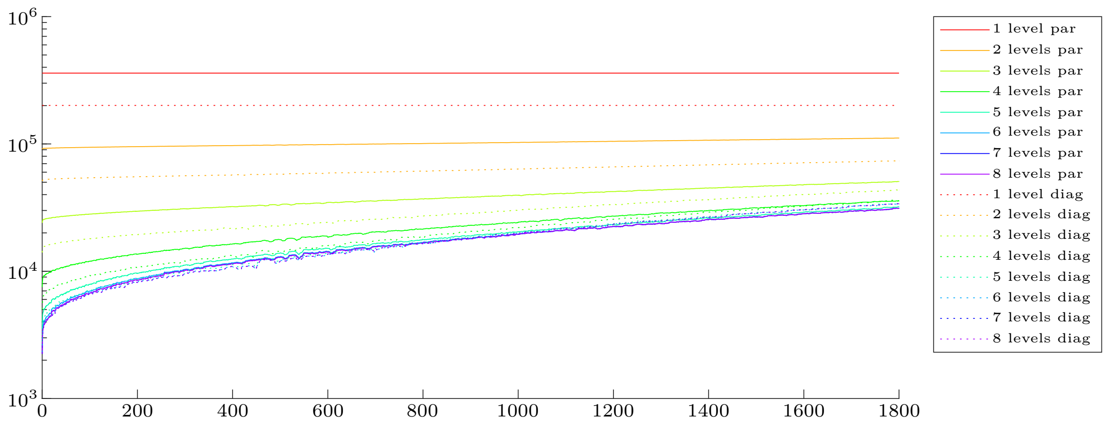
(b) tip velocity



(c) tip curvature



(d) arm curvature



(e) grid cells

Figure 7.13: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $\tau = 0.01$ for different number of levels, n_l, and minimal dilation parameters, $n_s = 1$ and $n_r = 0$.

artificial interfaces in the finest level continue to thin until it reaches n_s cell widths. When this limit is reached a regrid is triggered. In the general case, therefore, the thickness of the dilation layer ranges between $n_s + n_r$ and n_s.

For this reason the third subset of tests were performed for $n_r = 2, 4$ while keeping the $n_s = 1$ and $\tau = 0.0025$. Tip velocities and the errors in its computation are given in Tables 7.7 and 7.8 where also information about the total number of cells and regrids are given. It can be observed that specifying a regrid dilation parameter greater than zero reduces the occurrence of regrids by more than a third and similar improvements are recorded for both $n_r = 2$ and $n_r = 4$. In terms of total number of cells, an increase of about a sixth is observed when the regrid dilation is set to two cells while almost a 50% percent size increase is observed for $n_r = 4$. The first increment is comparable to the one observed in 7.7 for diagonal problems as the last one is with those in 7.8, even if in these cases the total with of the dilation layer is one cell wider (since $r_s = 1$) than in the previous two.

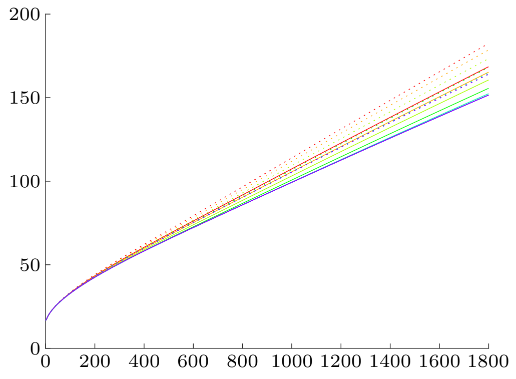
When comparing results obtained for different values of regrid–dilation parameter with the analogue obtained varying the stability–dilation parameter in terms of accuracy in the computation of the dendritic tip velocity, it is evident that a smaller reduction in the error is achieved by increasing n_r instead of n_s.

From the profiles provided in Figures 7.16 and 7.17, in particular from the curvature plots of the second one, some discontinuities are observed for $n_r = 4$. These are related to the fact that being $n_r + n_s \not< \texttt{MAX\_HALO\_DEPTH}$ there are issues in retrieving the phase field values around the tip required for computing the tip characteristics when the dendrite tip approaches the boundary between patches belonging to different processors.

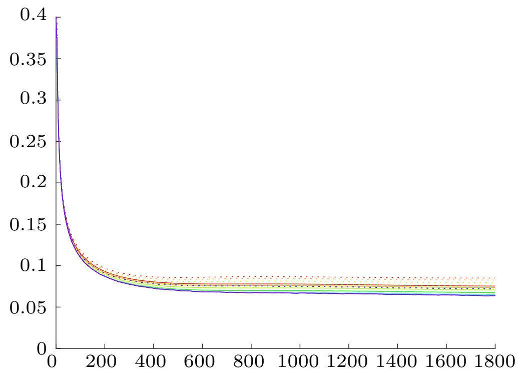
From the observations above, it can be concluded that in the case of PureMetal simulations using the PhaseField component, the best approach is to use the following set of regrid parameters

$$\tau = 0.0025, \qquad n_s = 4 \qquad n_r = 2\,. \tag{7.1}$$

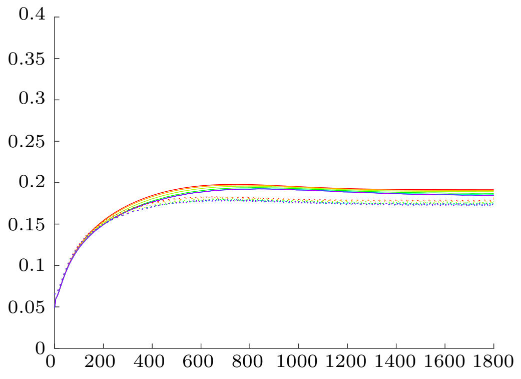
This choice of parameter requires to redefine the macro `MAX_HALO_DEPTH` to at least 7 but has the advantage to both lowering the error in computing the tip velocity and reducing the frequency of regrids, with the cost, however, of increasing the grid size more than necessary. This compromise could be improved in the future but either allowing different dilation parameters for each AMR level, or by adding a finer dilation parameter for enlarging the finest region immediately after the error estimate task is executed and before the current dilation algorithm is run by the regridder. The first approach needs to be implemented at a lower level within Uintah's Framework, wile the latter approch could be implemented both by the application developer as an additional task scheduled to run after the error estimate one, or the framework itself.



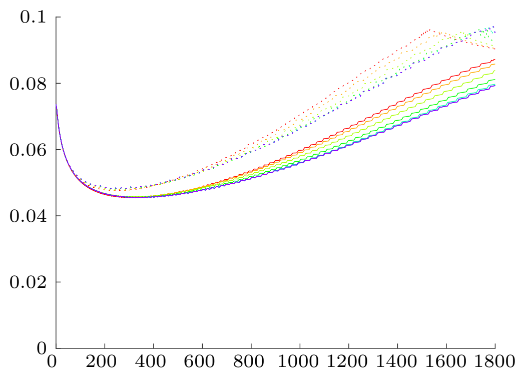
(a) tip position



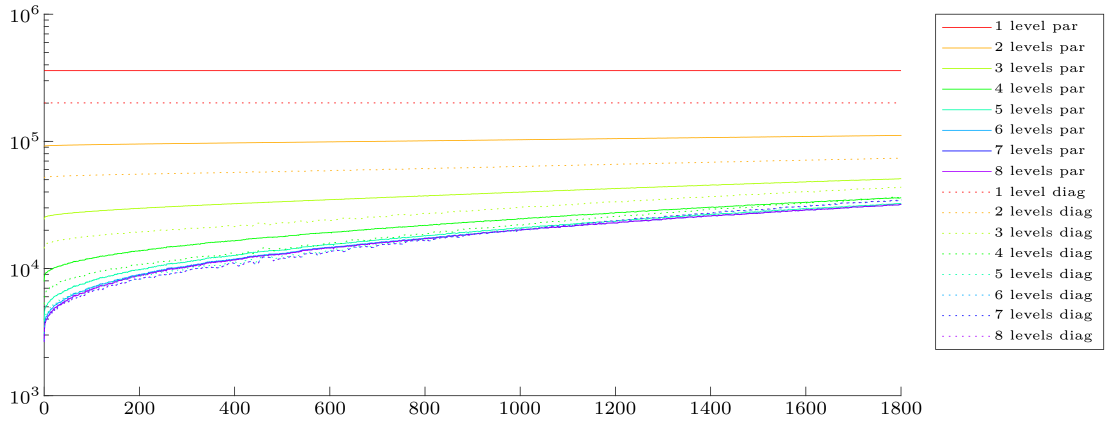
(b) tip velocity



(c) tip curvature

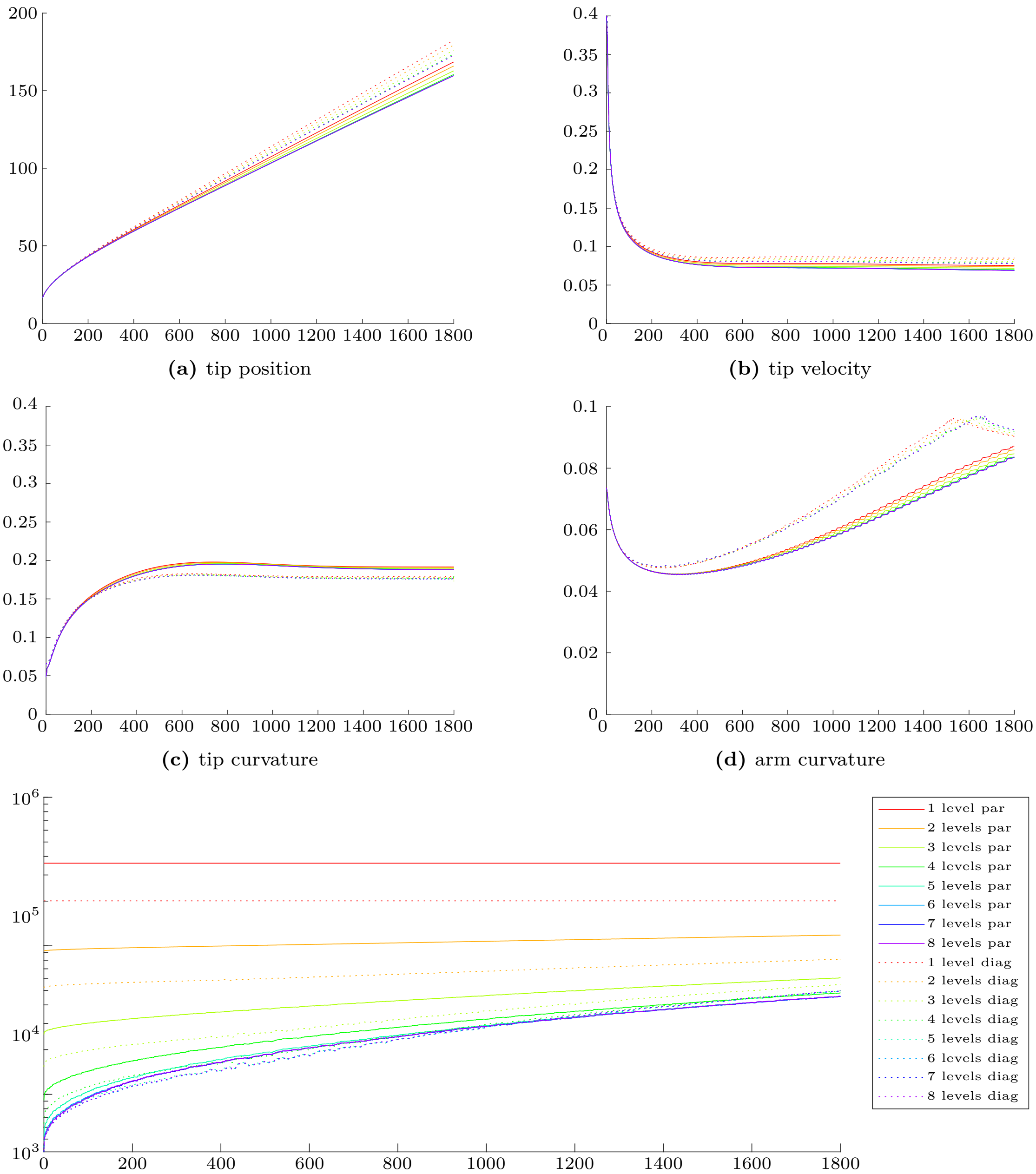


(d) arm curvature



(e) grid cells

Figure 7.14: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $n_s = 2$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_r = 0$.



(a) tip position

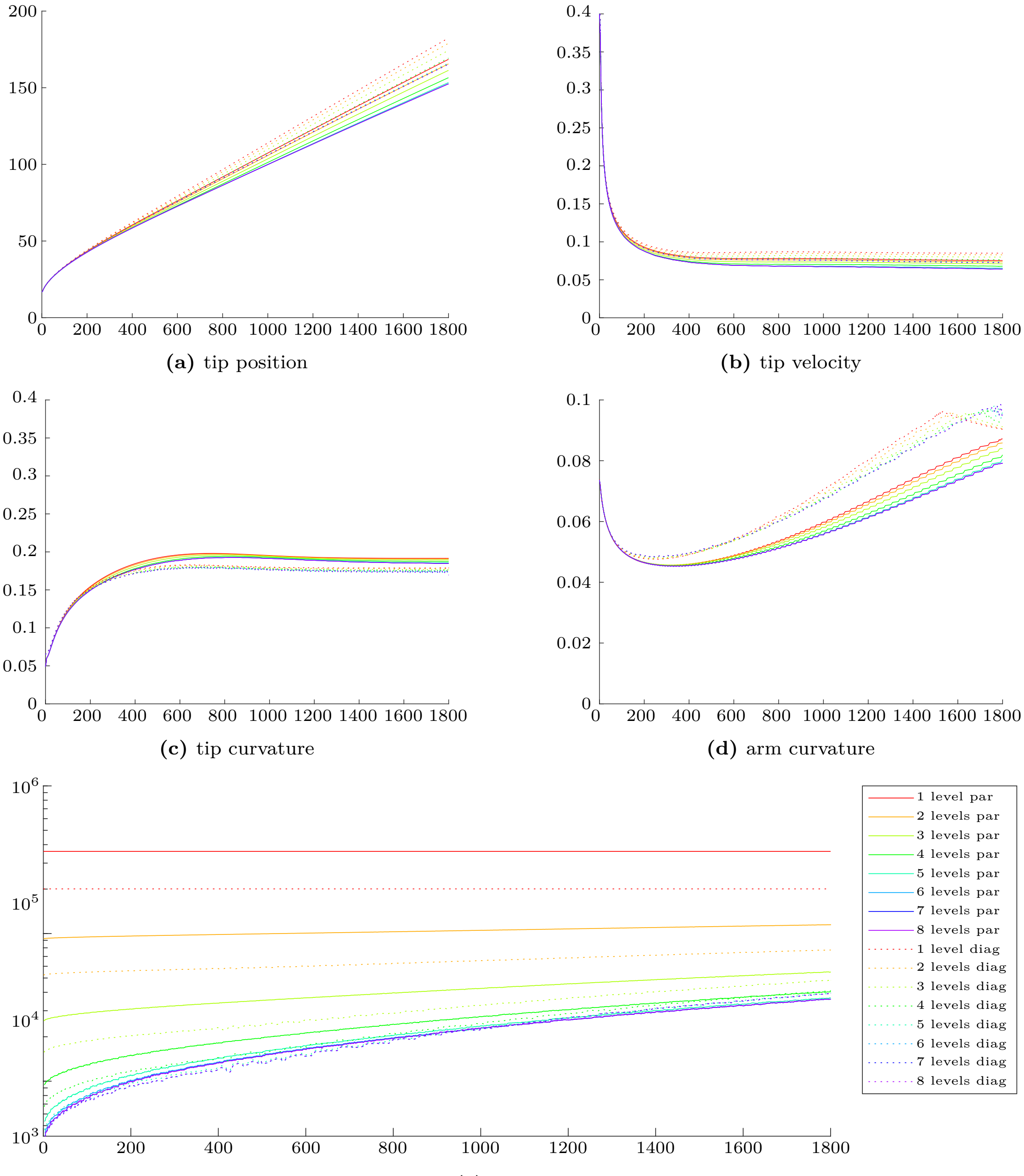
(b) tip velocity

(c) tip curvature

(d) arm curvature

(e) grid cells

Figure 7.15: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $n_s = 4$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_r = 0$.

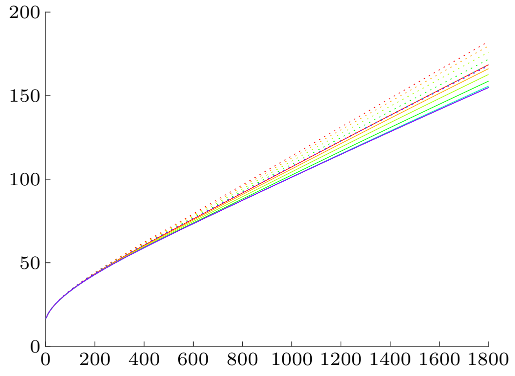


(a) tip position **(b)** tip velocity

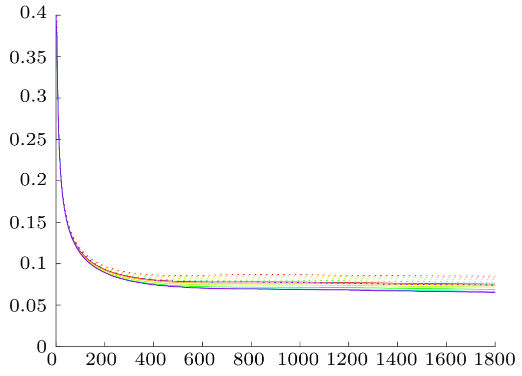
(c) tip curvature **(d)** arm curvature

(e) grid cells

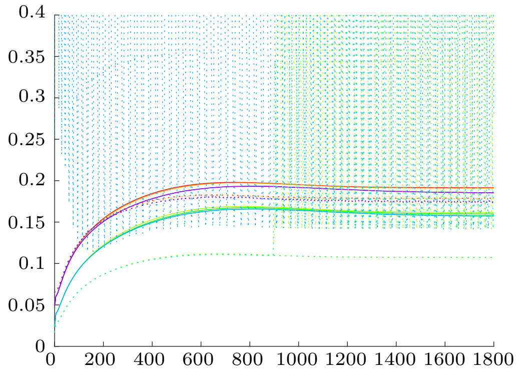
Figure 7.16: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $n_r = 2$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_s = 1$.



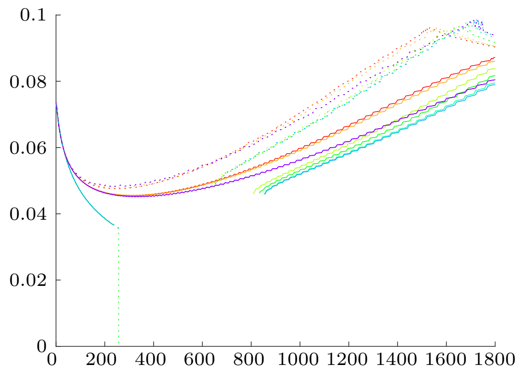
(a) tip position



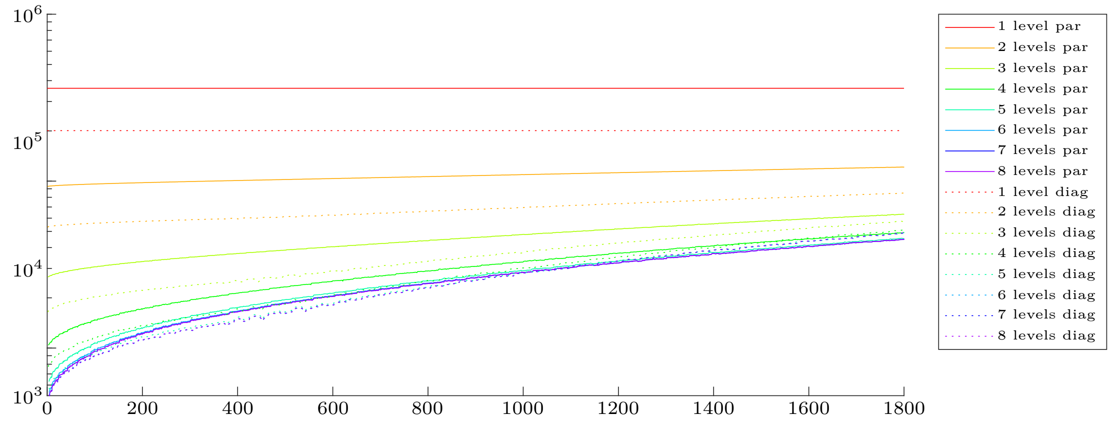
(b) tip velocity



(c) tip curvature



(d) arm curvature



(e) grid cells

Figure 7.17: Profiles of dendrite characteristics (a–d) and total number of grid cells (e) for PureMetal Problem I when $n_r = 4$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_s = 1$.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.00931	10.9%	1	360000	360000	360000
	2	0.07332	0.01133	13.3%	2927	101069	90960	109728
	3	0.07040	0.01424	16.8%	2791	37377	23988	48628
	4	0.06700	0.01765	20.8%	2708	21912	7648	33824
	5	0.06386	0.02079	24.5%	2608	18012	3476	30132
	6	0.06239	0.02226	26.2%	2563	17206	2512	29008
	7	0.06225	0.02240	26.4%	2571	17052	2276	28948
	8	0.06224	0.02240	26.4%	2569	17080	2276	29012
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08254	0.00210	2.4%	3079	61792	51136	72160
	3	0.07911	0.00554	6.5%	2969	27972	14032	41248
	4	0.07524	0.00941	11.1%	2853	19769	5040	33744
	5	0.07190	0.01275	15.0%	2737	17814	2832	31952
	6	0.07038	0.01427	16.8%	2718	17426	2372	31508
	7	0.07001	0.01464	17.2%	2700	17429	2304	31632
	8	0.07001	0.01464	17.2%	2700	17445	2320	31648

Table 7.3: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $\tau = 0.005$ for different number of levels, n_l, and minimal dilation parameters, $n_s = 1$ and $n_r = 0$.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.00931	10.9%	1	360000	360000	360000
	2	0.07338	0.01127	13.3%	2939	101948	90960	111376
	3	0.07062	0.01403	16.5%	2840	38472	23988	50756
	4	0.06723	0.01742	20.5%	2721	23055	7520	35744
	5	0.06441	0.02024	23.9%	2649	19167	3476	32228
	6	0.06282	0.02183	25.7%	2649	18365	2512	31296
	7	0.06263	0.02201	26.0%	2651	18212	2276	31236
	8	0.06263	0.02202	26.0%	2651	18241	2276	31300
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08265	0.00199	2.3%	3077	62747	51136	73648
	3	0.07929	0.00536	6.3%	2966	29158	14032	43552
	4	0.07566	0.00899	10.6%	2872	21010	5008	36160
	5	0.07221	0.01243	14.6%	2820	19061	2816	34096
	6	0.07087	0.01378	16.2%	2767	18680	2308	33764
	7	0.07049	0.01416	16.7%	2759	18684	2240	33920
	8	0.07049	0.01416	16.7%	2759	18700	2256	33936

Table 7.4: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $\tau = 0.0025$ for different number of levels, n_l, and minimal dilation parameters, $n_s = 1$ and $n_r = 0$.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.00931	10.9%	1	360000	360000	360000
	2	0.07342	0.01122	13.2%	2817	102151	91216	111472
	3	0.07068	0.01397	16.5%	2774	38724	24340	50884
	4	0.06730	0.01734	20.4%	2725	23347	7888	36032
	5	0.06485	0.01980	23.3%	2679	19547	3828	32580
	6	0.06388	0.02077	24.5%	2672	18807	2944	31984
	7	0.06378	0.02087	24.6%	2682	18671	2708	31956
	8	0.06378	0.02087	24.6%	2682	18703	2708	32020
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08270	0.00195	2.3%	3060	62743	51392	73920
	3	0.07934	0.00531	6.2%	3225	29169	14384	43680
	4	0.07580	0.00885	10.4%	3242	21082	5248	36480
	5	0.07298	0.01166	13.7%	3171	19254	3072	34736
	6	0.07201	0.01263	14.9%	3167	18970	2708	34596
	7	0.07188	0.01276	15.0%	3166	19008	2640	34656
	8	0.07188	0.01276	15.0%	3166	19024	2656	34672

Table 7.5: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $n_s = 2$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_r = 0$.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.00931	10.9%	1	360000	360000	360000
	2	0.07375	0.01089	12.8%	2792	106075	91856	118128
	3	0.07173	0.01292	15.2%	3296	44576	25300	61060
	4	0.06997	0.01467	17.3%	3316	30325	9120	48224
	5	0.06916	0.01549	18.3%	3387	27149	5204	45764
	6	0.06908	0.01556	18.3%	3386	26665	4432	45600
	7	0.06909	0.01556	18.3%	3386	26655	4276	45652
	8	0.06908	0.01556	18.3%	3387	26741	4276	45828
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08316	0.00149	1.7%	2980	66555	52032	81184
	3	0.08085	0.00380	4.4%	3485	34928	15344	54976
	4	0.07881	0.00584	6.9%	3563	27974	6352	49872
	5	0.07801	0.00664	7.8%	3570	26753	4384	49552
	6	0.07797	0.00668	7.8%	3567	26704	4100	49716
	7	0.07786	0.00679	8.0%	3577	26828	4112	50032
	8	0.07786	0.00679	8.0%	3577	26844	4128	50048

Table 7.6: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $n_s = 4$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_r = 0$.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.00931	10.9%	1	360000	360000	360000
	2	0.07360	0.01105	13.0%	1789	104079	91984	114752
	3	0.07102	0.01362	16.0%	1765	41132	25588	55012
	4	0.06794	0.01671	19.7%	1723	25841	9456	40400
	5	0.06543	0.01922	22.7%	1709	22022	5332	36708
	6	0.06436	0.02028	23.9%	1713	21278	4480	35952
	7	0.06425	0.02040	24.0%	1707	21143	4244	35924
	8	0.06425	0.02040	24.1%	1707	21176	4244	35988
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08292	0.00172	2.0%	1890	64618	52224	77408
	3	0.07995	0.00470	5.5%	1826	31511	15712	48272
	4	0.07642	0.00823	9.7%	1834	23473	6720	41056
	5	0.07373	0.01091	12.8%	1782	21629	4672	39584
	6	0.07277	0.01188	14.0%	1801	21340	4308	39412
	7	0.07263	0.01201	14.1%	1805	21380	4240	39504
	8	0.07263	0.01201	14.1%	1805	21395	4256	39520

Table 7.7: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $n_r = 2$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_s = 1$.

		tip velocity				total cells		
	levels	asymptotic	abs err	rel err	regrids	avg	min	max
parallel	1	0.07534	0.00931	10.9%	1	360000	360000	360000
	2	0.07391	0.01074	12.6%	1767	107978	92848	121456
	3	0.07174	0.01291	15.2%	1751	46031	26612	63428
	4	0.06902	0.01562	18.4%	1713	30989	10400	49296
	5	0.06660	0.01805	21.3%	1694	27186	6308	45476
	6	0.06557	0.01908	22.5%	1744	26436	5424	44912
	7	0.06545	0.01920	22.6%	1733	26303	5188	44820
	8	0.06545	0.01920	22.6%	1733	26336	5188	44884
diagonal	1	0.08505	0.00039	0.4%	1	200704	200704	200704
	2	0.08333	0.00131	1.5%	1849	68421	53024	84912
	3	0.08082	0.00383	4.5%	1815	36293	16432	57520
	4	0.07772	0.00693	8.1%	1781	28501	7728	51216
	5	0.07492	0.00973	11.4%	1792	26624	5712	48848
	6	0.07408	0.01057	12.4%	1793	26331	5252	49012
	7	0.07393	0.01072	12.6%	1783	26371	5184	49136
	8	0.07393	0.01072	12.6%	1783	26387	5200	49152

Table 7.8: Computed asymptotic velocities and total number of cells for PureMetal Problem I when $n_r = 4$ for different number of levels, n_l, and fixed $\tau = 0.0025$ and $n_s = 1$.

Chapter 8

PhaseField Uintah Component Implicit Solvers

In the previous section, support to adaptive mesh refinement has been introduced with the aim to reduce the computational grid size while maintaining the spatial refinement required to describe the phase field gradient at the physical interface between liquid and solid phases. In this section, support to an implicit solver is introduced to allow to overcome the limitations on the timestep size due to the stability constraint of the explicit Euler time scheme.

Implicit or semi-implicit time schemes are implemented in the PhaseField component for the Heat and Allen–Cahn Benchmark and PureMetal applications which require the solution of linear systems. For this purpose, the Uintah Computational Framework provides a driver to the HYPRE library [29], which features a collection of highly scalable solvers and preconditioners.

An analysis of the current status of Uintah's support of HYPRE solvers is given §8.1; two strategies are there presented for allowing the resolution of linear systems on semi-structured grids as those arising from AMR. The first allows to use the current Uintah's driver with fewer modifications to its implementations, while the second is the implementation of an additional driver to support HYPRE semi-structured solvers.

Details of the implementation of both strategies are available in §8.2.

To validate the two implementations, in §8.3 convergence analyses are presented for the Heat problem in 2D and 3D, and the Allen–Cahn benchmark, and for the PureMetal applications.

The two implementaions proposed here differ from other sotware implementations. For example, the AMReX [105] sotware, which offers probably one of the most complete and mature framework for block-structured adaptive mesh refinement (it replaces Chombo [21] and BoxLib [106]), follows the following approach. Starting from the finest AMR level a V-cycle is performed on each level, down to the coarsest one. Then either HYPRE, petsc, or some internal linear solver is then used to solve the problem on the full-coarsest level including the regions under refined block. A correction is eventually interpolated from the coarset to the finest AMR level. Theese steps are iterated until the norm of the residual on the finest level is below a given tolerance.

Both the smooth operator used by the V-cycle on refined levels and the solve operator on the coarsest level which assemble the linear matrix are implemented for each problem and for each variable basis (either at cell centers or nodes). At the moment of writing, AMReX implements solvers for two kind of problems: the first can be used for solving problems in the canonical form

$$(A\alpha - B\nabla\cdot\beta\nabla)\varphi = f\,,$$

where A and B are scalar constants, α and β are scalar fields, φ is the unknown, and f is the right-hand side of the equation; the second one for solving the Poisson's equation, which is a special case of the canonical form.

The approach proposed here differs from the other software, in using the external solver library HYPRE to update the solution on all the AMR levels. A second difference is that the solution is computed only on those portions of each level that are not refined any further, which reduces the overall problem size. The third and most beneficial novelty is the fact that the implicit solver implementation presented here decouples the definition of the problem from the linear solver implementation, making it possible to write new solver for new problems only by implementing new kernels instead of adding new solver implementations.

It has to be reminded that little focus has been given to them in this research work, even though the choice of the preconditioner greatly affects the performance of linear solvers. This research has focused more on the implementation of the software itself and only support for preconditioners has been developed. Which particular preconditioner is the most suitable for any of the applications considered is beyond the remit of this work.

8.1 Analysis

The HYPRE library offers four different conceptual interfaces to a collection of parallel multigrid solvers and preconditioners;

Structured Grid System Interface, which is suitable for applications over computational domains that can be described as unions of rectangular grids with the same grid step and a fixed stencil pattern of non-zeros at each grid point; this interface supports only a single unknown per grid point

Semi-Structured Grid System Interface, which is appropriate for applications whose domain can be described as union of structured grids as it is the case of composite grids arising from adaptive mesh refinement applications; this interface supports multiple unknowns per grid point

Finite Element Interface, which is convenient for linear problems arising from finite element discretization

Linear-Algebraic System Interface, which provides the traditional interface to matrices and vectors through row and column indices.

Within the UCF only the structured interface is fully supported for cell-centred variables by the `HypreSolver` driver. Only experimental support is provided for vertex-based variables with the structured interface. Unfortunately, this driver is not suitable for AMR simulations since the structured grid interface does not support the semi-structured grids on which such simulations are performed.

An old unmaintained driver is implemented which was developed to provide an interface to the semi-structured implementations within HYPRE. This `AMRSolver` driver, unfortunately, has never been working properly according to its developer and therefore its development has been abandoned.

Two different approaches have been identified for using HYPRE solvers on AMR semi-structure grids. The first consist in solving each AMR level separately using the already implemented structured `HypreSolver` driver. This is possible since each AMR level is a structured grid on its own. At each time step, first the problem on the coarsest level is solved. Since the coarsest grid coincides with the whole computational domain, its boundaries are only physical boundaries, therefore the physical boundary conditions are sufficient to compute the solution $\boldsymbol{x}^{N+1,0}$ given its values at the previous timestep $\boldsymbol{x}^{N,0}$. On refined levels, $\mathscr{T}_l^N$ $(l = 1, \dots, l_{\max})$, the grid boundaries may not be physical boundaries but artificial fine coarse interfaces. For this reason the solution $\boldsymbol{x}^{N+1,l-1}$ on the coarser level is used as Dirichlet condition on artificial boundaries to compute $\boldsymbol{x}^{N+1,l}$ from $\boldsymbol{x}^{N,l}$. This requires that the coarser level is solved before being able to solve the next refined level. Since the initialization of HYPRE structures is quite expensive, these may be reused across several timesteps. These structures depend on the level geometry and therefore a set of HYPRE structures is necessary for each level. Some tweaks have been necessary to allow having multiple sets of HYPRE structures coexist in the DataWarehouse as described in the next section. However these structures must be reinitialized after each regrid.

The second approach is to implement a driver to HYPRE's semi-structured interfaces. It has been decided to implement a new driver rather than fix the existing `AMRSolver` since it would have been easier to start from the structured working `HypreSolver` driver and then modify it in steps by adding a feature at a time from the `AMRSolver` fixing any bug and updating the interfaces to the latest release of the HYPRE library. Luckily, HYPRE's semi-structured grids are structured very similarly to Uintah's: Uintah's grid are organized in levels, each of which is a structured grid. Levels are then subdivided in patches, that are cubic clusters of cells. Patches are then distributed across MPI processes. Within HYPRE, the level counterpart is called *part* and patches are called boxes. The main difference between the two implementation, is that in Uintah the geometry is generated by the Regridder component and then the LoadBalancer divides the patches between processes, so that every MPI process is given the whole geometry and then works on only some patches of it. On the contrary in HYPRE, minimal global information is shared between processes and boxes are initialized only on the process owning it. This means that the initialization of HYPRE structures has to be delayed to after patches have been distributed, that is HYPRE must be initialized by tasks.

Another similarity between HYPRE and Uintah is the use of stencils. In Uintah the stencil is fixed to 5-points in two dimensions and 7-points in three while in HYPRE any stencil up to 27 points can be specified (1 layer around each point) and therefore the driver has to take care of mapping stencil values from Uintah's DataWharehouse to HYPRE matrices. In Uintah only matrices are exclusively represented as stencil-valued fields, therefore in the semi-structured cases where the matrix associated to the problem may have non-zero entries other than stencil entries, such as connecting cells between levels, this will require the introduction of new structure in the UCF to store these entries. Since we want to be able to initialize HYPRE structures only when required, these structures for handling the additional matrix entries must provide a mechanism for defining the non-zero matrix pattern while the HYPRE matrix structure is initialized and modifying their values at any timestep. Since extra matrix values in HYPRE are stored in a vector it is important that the structure introduced for handling extra entries stores the position of each entry within this vector to allow an efficient update of the matrices values at any given timestep.

The semi-structured driver is implemented here to support only one variable, but it can be extended to solve algebraic systems of multiple variables as supported by HYPRE's semi-structured interface. This will be necessary to solve fourth order problems such as the Cahn-Hilliard ones. The most straightforward strategy suitable to tackle this family of problem is to introduce an auxiliary variable $v = \Delta u$, since the discretization of the bi-Laplacian would require wider stencils not supported by HYPRE. The only interface supporting multiple variables in HYPRE is the semi-structured one. For these reason, the implementation of the new semi-structured driver is designed to support multiple variables at a later stage.

8.2 Implementation

As depicted in the previous analysis, two strategies are going to be implemented hereafter. The first one consists in using the existing `HypreSolver2` driver on each AMR level, and requires fewer modifications to run; the second one consists in implementing a new driver, `SStructSolver`, to be used on the whole AMR composite grid. These drivers implementation is described in the next two paragraphs. In the following paragraph, the list of modifications that have been necessary for the variable views introduced in the previous two sections are given. In the last paragraph of this section, an overview on how to integrate the drivers in a PhaseField application is given.

8.2.1 Structured Grid Solver

The typical workflow for using the `HypreSolver2` driver that is followed by Uintah's non-AMR applications on a structured grid is the following:

1. The `HypreSolver2` driver is instantiated by the Uintah's `SolverFacory` when sus's main is executed. The driver is then accessible to applications as a port through the `SolverInterface` interface.

2. Uintah's variables for storing the linear system to be solved are created within the application constructor. Grid variables of type `Stencil7` are used for the linear matrix in the general non-symmetric case and of type `double` for the solution, guess and right–hand–side vectors.

3. The `SolverInterface`'s `readParameters` method is invoked by the application `problemSetup` method to parse the input file and initialize an instance of the class `HypreParams`, which specializes `SolverParameters` class, and will be later used to select which HYPRE solver to use and its configuration. This step is executed by the `SimulationController` immediately after all components are initialized by sus.

4. When the application initialization task is scheduled at the first (with its `scheduleInitialize` method) or at a restart timestep (with `scheduleRestartInitialize`), the application also schedules the solver's `initialize` task by invoking its `scheduleInitialize` method. This task is specified to be of type `OncePerProc` so that it will then executed by each available process exactly once.

5. When the application `scheduleTimeAdvance` method is executed to schedule the task that advances the solution to the next timestep, it invokes the solver's `scheduleSolve` method. This method first creates an instance of the class template `HypreStencil7` whose purpose is to store all information about which `VarLabels` from which `DataWarehouse` must be used to populate the values of the structures required to solve a linear system within HYPRE. This class is templated upon the type of variable basis used in Uintah (cell-, face- or vertex-based) while for HYPRE the cell-centered base is always used, since it is the most widely supported. To comply with this changes Patches ranges must be adjusted when later they are converted to boxes in the HYPRE structured grid. The `scheduleSolve` then schedules the `HypreStencil7`'s `solve` task, whose `TaskType` is `OncePerProc`.

6. After the `TaskGraph` is compiled, the `Scheduler` ensures that the solver `initialize` task is executed by each process once per process. This task allocates all HYPRE's structures: the matrix, vectors and solver structures that constitute the members of the `hypre_solver_struct` class which is then saved in the `DataWarehouse` as a `SoleVariable` so that the same instance is shared by all patches belonging to the same processor.

7. At each timestep, the `Scheduler` controls the execution of the `HypreStencil7`'s `solve` task. This is executed once by each available process, and it is within this task that HYPRE routines are invoked to actually populate HYPRE structures, assemble and solve the linear system associated to the application. The following steps are performed by the `solve` task:

Structures Retrieval — The `hypre_solver_struct` containing all HYPRE instances is retrieved from the `DataWharehouse`.

(a) First the current DataWharehouse is searched and if the structure is not found this is moved from the old to the current. This check has to be performed because there are applications that may use the same solver more than once per timestep and the solver structure may have been already moved from the old DataWarehouse.

Grid and Stencil Setup — If the current timestep is a *setup timestep* – i.e. it is the first one at the beginning of a simulation or after it has been restarted or if the number of timestep since last setup coincides with the frequency the user has specified with the parameter `setupFrequency` in `Solver/Parameters` node of the ups file – the following setup and assemble steps are performed:

(b) `HYPRE_StructGrid` is created (or deleted and recreated). For each patch belonging to the processor the function `HYPRE_SetExtents` is used to specify the geometry of the boxes for HYPRE's structured grid and `HYPRE_SetPeriodic` is called if Uintah's grid is periodic in any direction. Finally the grid is assembled with `HYPRE_StructGridAssemble`.

(c) `HYPRE_Stencil` is (re)created and its elements are populated with the offset vectors corresponding to the members of the Uintah's `Stencil7` class calling `HYPRE_StructStencilSetElement`. `Stencil7` are mapped to offsets in the following order: `p` $\mapsto$ `{0,0,0}`, `e` $\mapsto$ `{1,0,0}`, `w` $\mapsto$ `{-1,0,0}`, `n` $\mapsto$ `{0,1,0}`, `s` $\mapsto$ `{0,-1,0}`, `t` $\mapsto$ `{0,0,1}`, `b` $\mapsto$ `{0,0,-1}`. Nonsymmetric problems are treated in HYPRE, as in all other Uintah's components, as three dimensional problems regardless of the physical dimension of the problem. For cell centered problems, as those considered here, this approach does not affect the size of the linear system in HYPRE.

Matrix Setup, Update and Assemble — Matrix is created, its entries are set/updated and the matrix finalized.

(d) `HYPRE_StructMatrix` is (re)created from the given grid and stencil and its properties (symmetry and number of ghosts) are set. Finally the matrix is initialized via the `HYPRE_StructMatrixInitialize` function.

(e) If the current timestep is a setup timestep or if the number of timesteps since last update coincides with the frequency the user has specified with the parameter `updateCoefFrequency` in `Solver/Parameters` node of the ups file, the matrix entries are set using the grid `Stencil7` values from a given variable from the `DataWarehouse` . Values from the `Stencil7` grid values of `A`, in particular, are rearranged to match the order of offsets in the `HYPRE_StructStencil` object.

(f) If the current timestep is a setup timestep the HYPRE matrix is assembled calling the `HYPRE_StructMatrixAssemble` routine.

RHS Vector Setup, Update and Assemble — The right-hand-side vector and the solution vector is created, its entries are set/updated and the vector is finalized. These steps are grouped in `HypreStencil7`'s method `createPopulateHypreVector`

(g) If the current is a setup timestep an `HYPRE_StructVector` is (re)created and initialized.

(h) Its coefficients are always updated, contrarily to the matrix ones. Analogously to the matrix, however, values are filled by lines of cells. In this case there is no need to rearrange the values from the grid variables since values corresponding to a line are adjacent within the Uintah's grid variable associated to a vector.

(i) If the current timestep is a setup timestep the HYPRE vector is assembled calling the `HYPRE_StructVectorAssemble` routine.

Solution Vector Setup, Update and Assemble — The solution vector is created, its entries are set/updated and the vector is finalized. When specified, this vector can also be used as the initial guess by HYPRE's iterative solvers.

`HypreStencil7`'s method `createPopulateHypreVector` is invoked so that steps (g–i) are performed for the solution/guess vector.

System Resolution – A switch clause is used to switch between different HYPRE solver implementations and the following steps are performed

(j) It the current is a setup timestep, a solver object is (re)created and setup according to the parameters specified by the user in input. The solver instance is saved as a member of the `hypre_solver_struct` object which is kept in the `DataWarehouse` a `SoleVariable`.

(k) It the current is a setup timestep and a preconditioner has been selected in the user input, a preconditioner object is (re)created, setup and passed to the solver. The preconditioner instance is saved as a member of the `hypre_solver_struct` object.

(l) The linear system is solved using the appropriate HYPRE solve routine.

(m) The number of performed iterations and the final relative residual norm are retrieved and a convergence check is performed.

(n) The values of the solution are copied back from the HYPRE˙StructVector object to Uintah's grid variable.

(o) If the current is a setup timestep, the HYPRE grid and stencil objects are destroyed, since they are no longer needed after the matrix and vector have been assembled.

(p) If the convergence check is unsuccessful, a `ConvergenceFailure` exception is thrown to be caught by the `SimulationController`.

8. After the last timestep, the `hypre_solver_struct` object is destroyed and with it all instantiated HYPRE structures left are destroyed as well.

The following modifications have been necessary to allow the use of multiple `hypre_solver_struct` instances for solving one linear system for each AMR level. Since each grid level is executed individually by Uintah's scheduler, the tasks described before of the type `OncePerProc`, are actually executed by each process once per level. This means that when a solver task is executed all the patches within the same execution of a task belong to the same level, and that it can happen that, for a particular level, there are no patches assigned to a specific process. This edge case is assured to not happen by the load balancer on structured grids, since the condition that the number of total patches must be at least equal to the number of available processes is enforced. Some minor checks had to been added to correctly handle this case.

Apart from these small changes, the most significant modification concerned the use of multiple `hypre_solver_struct` instances, one for each AMR level. First, it has been necessary to associate a different `VarLabel` to each level's `hypre_solver_struct` instance. Secondly it has been necessary to modify step 7(a) of the previous workflow. In fact, it is now possible that the `hypre_solver_struct` relative to a level is not available in either the current nor the previous `DataWarehouse`. This may happen if after a regrid a level has been added. In this case a new instance of the `hypre_solver_struct` class is created and initialized similarly to step (6).

In addition to the previously available HYPRE structured solver for non-symmetric problems (GMRES, PFMG, SMG), support has been added also to the following: CycRed, BiCGSTAB, LGMRES, FlexGMRES, and Hybrid[1]. All of these solvers — except for the CycRed which is an exact solver — can be used also as preconditioners. For details about these solvers please refer to §2.3.3 and to the references therein.

8.2.2 Semi Structured Grid Solver

The biggest difference between solving multiple linear systems on each AMR level and solving a single global system for the whole AMR grid is that, in the first case, each system has no non-zero entries other than stencil entries while, in the latter, at fine-coarse interfaces additional non-zero entries are required.

For these reason the `HypreSStruct::AdditionalEntries` structure has been introduced to store these additional entries in Uintah's `DataWharehouses` as the `Stencil7` structure is used to store

[1]The Hybrid implementation was already available but supported only PCG as Krylov solver. The previous implementation has been modified to allow using GMRES as well

stencil ones.

The HypreSStruct::AdditionalEntries class maps keys of type HypreSStruct::MatrixIndex to double values. The HypreSStruct::MatrixIndex class has been defined to represent additional entries in the linear matrix that are not accessible via a cell-index and a stencil entry. The associated value is the coefficient of the linear matrix corresponding to a key.

HypreSStruct::AdditionalEntries key-values pair types can be passed to its the compound assignment by sum operator, +=, allowing an agile syntax for setting non-stencil entries from kernels of PhaseField applications. This operator either adds the entry to the map or, if its key is found in the map, updates its associated value with the given one.

For structured geometries in HYPRE it is sufficient to add to a HYPRE_StructGrid instance all its boxes (Uintah's patches). For semi-structured grids, however, it is necessary to add to a HYPRE_SStructGrid instance all its parts (Uintah's levels) and to each part its boxes. However, defining a grid is not sufficient because it is necessary to define also the connections between each part. The definition of the grid and, in particular, of the graph is not trivial but is common to all semi-structured solvers, therefore the information and functions introduced to this purpose have been defined as members and methods of the HypreSStruct::detail::sstruct_implementation template class. The template parameters are the problem dimension, DIM and the type of coarse to fine interpolation to use, C2F. The first template parameter allows to optimize some implementations while the second one is used to specify the order of interpolation to use when a connection is created from a coarse patch to a fine patch. Details of these implementations are given later in [18].

The HypreSStruct::detail::sstruct_implementation class partially corresponds to the HypreStencil7 class of the structured driver. However this class only contains those steps of the HYPRE workflow that are independent on the choice of the solver. In fact, the factory design pattern is used instead of the switch clause to select the implementation requested at runtime by the user.

The list of available solver and preconditioner semi-structured from HYPRE is given in Table 8.1

Application developers can access the semi-structered implementations through the HypreSStruct::SStructInterface abstract interface. This interface provides for the following pure virtual functions:

globalData — to retrieve all the information about the grid that is independent on the distribution of boxes (patches) to processors.

partData/isPartSet/setPart — to retrieve, to check if set or to set the HypreSStruct::PartData object that contains the information about a given part of the structured grid.

addBox — to assign a box to the current processor; its arguments are the part index, the patch id within Uintah, and the triplets with the bounding indices of box.

Implementation	S	P
SysPFMG	1	
Split	−2	
FAC	−3	
Maxwell	−4	
PCG	5	
GMRES	6	
FlexGMRES	7	
LGMRES	8	
BiCGSTAB	9	
None	–	0
Diagonal	–	−1

Table 8.1: Semi-structured solver and preconditioner implementations from HYPRE. On the right columns their underlying values in the `HypreSStruct::S` and `HypreSStruct::P` enumerations. Last two entries are not proper solver implementations but have been added to the `HypreSStruct::P` enumeration to count for the case where none or a diagonal scale precondition is used.

`gridInitialize` — to execute the step corresponding to (7.b) of the structured grid workflow. It creates the `HYPRE_SStructGrid` instance, assigns boxes, and set the grid variables and periodicity. Its only argument is the MPI communicator since all information required by the step should be previously provided via the constructor and `setPart`

`stencilInitialize` — to create and initialize the `HYPRE_SStructStencil` instance. This method corresponds to step (7.c) of the structured grid workflow. Its only argument is the MPI communicator since all information required to create the HYPRE stencil structure and to populate its entries are the problem dimension and the stencil offsets which depend only on the problem dimension, which is a template parameter of the `HypreSStruct::SStructInterface` class.

`graphInitialize` — to initialize the `HYPRE_SStructGraph` object. This step is the most complex addition to the structured grid workflow. In addition to the MPI communicator, an handle to Uintah's grid is requested as an input parameter together with an 3D-array of pointers to `HypreSStruct::AdditionalEntries`. This array is meant to provide the information about which entries must be added to the semi-structured graph for each variable, level/part and patch/box whose indices are those identifying the entries in the input 3D-array.

`matrixInitialize` — to create and initialize a `HYPRE_SStructMatrix` instance. This method corresponds to step (7.d) of the structured grid workflow. Its only argument is the MPI communicator since all information required to create and initialize the HYPRE matrix structure is the semi-structured graph which is supposed to be already initialized.

`rhsInitialize`/`solutionInitialize` — to execute the step corresponding to (7.g) of the structured grid workflow for either initializing the right-hand-side or the solution vector. Their only argument is the MPI communicator since all information required to create and initialize an HYPRE semi-strucured vector structure is the semi-structured grid which is supposed to be already initialized.

`solverInitialize` — to create and set the solver implementation parameters. This method corresponds to step (7.j) of the structured grid workflow but does includes the part where Uintah's data wharehouse is accessed. Implementations of the `HypreSStruct::SStructInterface` interface, in fact, are designed to provide implementations specific to the HYPRE workflow for solving a linear system on semi-structured AMR grids. Uintah's specific workflow is implemented in the `HypreSStruct::Solver` template class which will be discussed later in this subsection.

`matrixUpdate` — to execute the step corresponding to (7.e) of the structured grid workflow which updates the coefficients of the linear matrix. Its parameters are two 3D arrays, `stencil_entries` and `additional_entries`, whose elements are instances of `Stencil7` Uintah's grid variables, for the first array, and pointers to `HypreSStruct::AdditionalEntries`, for the second one. These are indexed, as for the `graphInitialize` argument, by the variable, level/part and patch/box indices.

`rhsUpdate/guessUpdate` — to update the coefficient of either the right-hand-side or the solution vector. These methods correspond to step (7.h) of the structured grid workflow. Their input argument is a 3D array whose elements are instances of `double` Uintah's grid variables, which are used to populate HYPRE's vector coefficients. Again, this array is indexed by the variable, level/part and patch/box indices.

`assemble` — to execute steps (7.f) and (7.i) of the structured grid workflow. This method tells HYPRE to assemble all the elements of the linear systems. It has no arguments.

`solverUpdate` — to setup the solver and, if necessary, the preconditioner in HYPRE. This method is equivalent to steps (7.j) and (7.k) and requires no arguments.

`solve` — to make HYPRE solve the linear system. As only parameter this methods requires a pointer to an instance of the structure `HypreSStruct::SolverOutput` where information about the resolution are saved: in particular the number of iterations and the Euclidean norm of the solution. This method corresponds to steps (7.l) and (7.m) of the structured grid workflow.

`getSolution` — to execute step (7.n) of the structured grid workflow. This method updates the given instance of `double` grid variable in input with the values of the solution vector computed by HYPRE solver.

`finalize/restart` — to free resources and reset the status of the HYPRE workflow. The first method is meant to be called at the end of a simulation while the latter should be called when, during the simulation, all structures must be reinitialized either because a regrid has occurred or because the user has specified so. These methods take no arguments and correspond to step (7.o).

For all semi-strucured solvers in Table 8.1 but Maxwell (which is as solver specific for the resolution of Maxwell equations) a partial template specialization has been defined. These specializations, in addition to the implementation of the virtual function required to complete the `HypreSStruct::SStructInterface`, whose implementations are straightforward, provide implementation independent names to access to the HYPRE routines that must be used to set a preconditioner, setup the solver and solve the assembled linear system with the chosen semi-structured solver implementation.

Only the `graphInitialize` and `matrixUpdate` method definitions are not straightforward. These two methods are the only ones that have to handle the semi-structured graph.

When the grid, the stencil and the graph are then used by HYPRE for initializing the `HYPRE_StructMatrix`, for each row a vector is reserved to contain as many values as entries in the stencil and in the graph. The first values are the matrix coefficients corresponding to stencil entries and the following correspond to the entries in the graph. Their order determines to which entry of the graph they correspond and when the matrix values are set or updated these are identified only by their index, therefore in the implementation of the semi-structured driver it is necessary to provide for an efficient mechanism to store the position in the matrix values arrays corresponding to additional entries.

Another issue related to the population of the `HYPRE_SStructGraph` entries is the fact that multiple contributions can affect the same graph entry. For example, at the corner of a refined patch which has its two adjacent edges lying on a fine-coarse interface, the interpolation of both the two virtual fine cells involves the same coarse cell opposite to that corner.

For this reason the `SStructInterface` contains two lists for storing additional entries: `extra_stn_entries` and `extra_add_entries`. The first list stores the information about *connections* that affect stencil values in a matrix row but that are not directly associated to an entry in the `Stencil7` grid variable associated to the matrix in Uintah, while the second list is used determine the additional values in a matrix row. To the first container, for example, belong entries for setting to zero stencil entries that cross a fine-coarse interface, but also more complex contributions arising from the interpolation/restriction operations.

Thanks to these containers the implementation of the `matrixUpdate` method, which is listed and described in detail in [18], is much simpler.

This method loops over all parts, boxes and variables indices to set the stencil matrix values, analogously to its structured counterpart described earlier. Here, however, in addition to setting the values using the `Stencil7` grid variables, both the `extra_stn_entries` and `extra_add_entries` containers' elements are traversed for each part and variable index to update the matrix-non stencil coefficients and to modify any stencil coefficient that may need to be updated.

The last step of the `matrixUpdate` method is ensuring that: (1) all stencil entries that point to ghost cells (because across a fine-coarse interface) are zeroed out; (2) all rows corresponding to cells that have been refined are zeroed with the exception of their diagonal entry. For this purpose routines provided for the semi structured FAC solver can be used because of their general implementation.

The containers that make this implementation simple `extra_stn_entries` and `extra_add_entries` are popuplated by the `graphInitialize` listed in [18].

The graph structure is created, configured and the stencil is set analogously to the structered case. Then the non stencil part of the graph is setup. First, entries connecting fine cells to coarse cells are added to the graph, then coarse to fine cells entries. In both cases the outer loop is performed over the fine patches, since in Uintah there is no direct way to check if a coarse cell has been refined by the `Regridder`, but it is immediate to retrieve which patches and cell ranges are covered by a refined patch on its coarser level.

Two type of coarse to fine connections are created, the first one is an artificial connection between a coarse cell and its refined one because in the FAC implementation, HYPRE reassigns coarse boxes to the available processors so that interpolation and restriction operations can be executed with fewer MPI communications, in particular it tries to assign coarse boxes to the same processor that owns its refined boxes. This reassignment, however, can fail if there is no entry in the matrix that connects the two original and the new processes. The introduction of this additional entry of zero-value in the matrix forces the communication of some information about the matrix structure and geometry before the reassignment is performed and prevents the FAC implementation from failing.

The second type of coarse to fine entries that are to be added to the graph are those connecting coarse cells to adjacent fine cells across a fine–coarse interface. After coarse to fine connections have been processed, the `updateMatrix` method moves to the fine to coarse connections. These are not determined by the geometrical collocation of fine patches with respect to coarser ones, but they are determined by the entries in the Uintah's `HypreSStruct::AdditionalEntries` variable instance. Before adding a fine to coarse connection, however, some checks must be performed. A `MatrixEntry` is, in fact, created when a differential operator of a view over a Uintah variable is called at a location on a fine-coarse interface. For their design, views are ignorant of any geometry information outside of the patches their instantiated on, and therefore it may happen that an additional entry may connect a fine cell to a coarse ghost cell that is actually being refined. In this case fine-to-fine entries arte pushed back to one of the extra container To select the right container to which to add each fine-to-fine entry, a check must be performed to see if each fine-to-fine entry correspond to a stencil entry. Otherwise, an entry is added to the `extra_add_entries` container and eventually an entry to HYPRE's graph is added. At the end of the `matrixUpdate` method, HYPRE's graph is eventually assembled.

8.2.3 Views

In this paragraph the modification to the view structures introduced in §5.2 and extended in §6.2 for supporting both implicit drivers is given.

The idea is to provide alternative implementation to differential operators to the ones implemented for the explicit time stepping scheme. The Laplacian method should return at each point the values of the stencil entries of the matrix and of the contribution to the corresponding entry of the right-hand-side vector of the linear system, for the structured solver, or, in addition to these, also the non-stencil additional entries of the matrix for the semi-structured solver. Moreover, since for some applications the linear matrix is independent on the timestep, additional implementation should be provided for computing only the contributions on the right-hand-side vector.

For this reason the following additional methods are defined for the `PhaseField::DWFDView` in its public base class `PhaseField::detail::dw_fd_view<ScalarField<T>`: `laplacian_sys_hypre`, `laplacian_rhs_hypre`, `laplacian_sys_hypresstruct`, and `laplacian_rhs_hypresstruct`.

Methods with the `hypre` suffix are meant to be used with the `HypreSolver2` driver while those with the the `hypresstruct` suffix are meant to be used with the `HypreSStruct` driver. The `sys` suffix identifies the methods which return both matrix and rhs entries while the `rhs` one identifies those which return only the rhs contributions.

These methods forward to the methods which are templetazed on the direction along which the second order derivative is approximated. Dynamic polymorphism is used to provide for the implementation appropriate to the particular grid partition (see §5.2) and take into account any boundary and fine-coarse conditions.

For internal regions, the second order derivatives influence only stencil entries, therefore `rhs` implementations need do nothing.

Boundary views, instead, contribute to both stencil entries of the matrix and to the right-hand-side vector. For example, the `PhaseField::detail::bc_fd` template specialization for cell-centered `ScalarField` and 1-cell offset stencils, on faces `F` where a `BC::Neumann` boundary condition is enforced, provides for the following implementations:

```
/// Boundary face normal vector direction
static constexpr DirType D = get_face<F>::dir;

/// Boundary face normal vector sign (int)
static constexpr int SGN = get_face<F>::sgn;

/// Boundary face normal vector sign (double)
static constexpr double DSGN = get_face<F>::dsgn;

template < DirType DIR >
inline typename std::enable_if < D == DIR, void >::type
```

```
add_d2_sys_hypre ( const IntVector &, S & stencil_entries, V & rhs ) const
{
  double h2 = m_h[D] * m_h[D];
  stencil_entries[F - SGN] += 1. / h2;
  stencil_entries.p += -1. / h2;
  rhs += DSGN * m_h[D] * m_value;
}

template < DirType DIR >
inline typename std::enable_if < D == DIR, void >::type
add_d2_rhs_hypre ( const IntVector &, V & rhs ) const
{
  rhs += DSGN * m_value;
}

template < DirType DIR >
inline typename std::enable_if < D == DIR, void >::type
add_d2_sys_hypresstruct ( const IntVector &, S & stencil_entries, A &, V & rhs ) const
{
  double h2 = m_h[D] * m_h[D];
  stencil_entries[F - SGN] += 1. / h2;
  stencil_entries.p += -1. / h2;
  rhs += DSGN * m_h[D] * m_value;
}

template < DirType DIR >
inline typename std::enable_if < D == DIR, void >::type
add_d2_rhs_hypresstruct ( const IntVector &, V & rhs ) const
{
  rhs += DSGN * m_value;
}
```

Additional non-stencil entries are created only at fine-coarse interfaces since it is only at these interfaces that the stencil connects two different parts of the semi-structured AMR grid. In [18] are listed the implementations of the second order derivative operator for this case together with a detailed description of the code.

8.2.4 Applications

Two implicit time-stepping methods have been considered: the Backward Euler and the Crank–Nicolson. These methods have been implemented for the Heat application and for the other applications, whose problem is nonlinear, they have been applied to as semi-implicit schemes where the nonlinear terms of the problem have been evaluated at the previous timestep. The derivation of the linear systems resulting from these methods for the heat problem has already been discussed in §2.3.2.

The Allen–Chan semi-discrete equation corresponding to 2.11 is, for the Backward Euler scheme,

$$\frac{u^{N+1}-u^N}{k} = \varepsilon^2 \Delta u^{N+1} - [(u^N)^2+1]u^{N+1}, \qquad \text{on } \Omega\,,$$

or, for the Crank–Nicolson scheme,

$$\frac{u^{N+1}-u^N}{k} = \varepsilon^2 \Delta \frac{u^{N+1}+u^N}{2} - [(u^N)^2+1]\frac{u^{N+1}+u^N}{2}, \qquad \text{on } \Omega\,.$$

When finite differences are used to discretize the equation in space, the following linear system is obtained for the first scheme

$$[\mathbb{1} - k\mathbb{B}^N + k\varepsilon^2\Delta]\boldsymbol{u}^{N+1} = \boldsymbol{u}^N + \boldsymbol{b}^N, \qquad N = 1,2,\dots\,,$$

and the following for the second one

$$[\mathbb{1} - \tfrac{k}{2}\mathbb{B}^N + \tfrac{k}{2}\varepsilon^2\Delta]\boldsymbol{u}^{N+1} = [\mathbb{1} + \tfrac{k}{2}\mathbb{B}^N - \tfrac{k}{2}\varepsilon^2\Delta]\boldsymbol{u}^N + \boldsymbol{b}^N, \qquad N = 1,2,\dots\,,$$

where $\mathbb{1} \in \mathbb{R}^{M\times M}$ is the identity matrix, $\Delta \in \mathbb{R}^{M\times M}$ is the matrix arising from the discretization of the Laplacian, and $\mathbb{B}^N \in \mathbb{R}^{M\times M} : B^N_{ij} := \delta_{ij}[(u^N_i)^2 - 1]$ is the diagonal matrix corresponding to the non-linear part of the equation, and the vector $\boldsymbol{b}^N$ arises from the imposition of boundary conditions. While Δ is constant over the simulation (at least when no regrid is performed), the matrix $\mathbb{B}^N$ depends on the timestep, therefore, contrarily to the heat application, the linear matrix for this application must be updated at every timestep. The matrix Δ is the same matrix arising from the discretization of the heat problem, and depends on the implicit solver driver being used. When a structured solver is used for each AMR level, this matrix can be written as a lower triangular block matrix with the first diagonal block corresponding to the coarsest level. These schemes have been implemented in the `Benchmark02` application.

No semi-implicit time scheme has been yet implemented for the other benchmark applications, because Cahn-Hilliard problems (2.15) require, for the discretization of the bi-Laplacian, either a larger stencil, which is not supported by HYPRE, or to introduce an auxiliary variable $v = \Delta u$, so that at every time step a system of two second order equations has to be solved, but multiple variables is not supported by the structured interface and not yet implemented in the semi-structured driver. However, in the second approach, the Backward-Euler semi-implicit scheme will result in the following semi-discrete system of equations

$$\begin{cases} u^{N+1} &= u^N - k\Delta v^{N+1} \\ v^{N+1} &= \varepsilon^2\Delta u^{N+1} - [(u^N)^2 - 1]u^{N+1} \end{cases}, \qquad \text{on } \Omega\,,$$

and the Crank-Nicolson scheme will result in the following semi-discrete system of equations

$$\begin{cases} u^{N+1} &= u^N - \frac{k}{2}\Delta v^{N+1} - \frac{k}{2}\Delta v^N \\ v^{N+1} &= \frac{\varepsilon^2}{2}\Delta u^{N+1} + \frac{\varepsilon^2}{2}\Delta u^N - \frac{(u^N)^2-1}{2} u^{N+1} - \frac{(u^N)^2-1}{2} u^N \end{cases}, \qquad \text{on } \Omega\,.$$

The associated fully-discrete linear system is, for the first method

$$\begin{bmatrix} \mathbb{1} & k\Delta \\ -\varepsilon^2\Delta + \mathbb{B}^N & \mathbb{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^{N+1} \\ \boldsymbol{v}^{N+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}^N \\ \boldsymbol{0} \end{bmatrix} + \boldsymbol{b}^N, \qquad N = 1, 2, \dots\,,$$

and, for the Crank-Nicolson scheme it is

$$\begin{bmatrix} \mathbb{1} & \frac{k}{2}\Delta \\ -\frac{\varepsilon^2}{2}\Delta + \frac{1}{2}\mathbb{B}^N & \mathbb{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^{N+1} \\ \boldsymbol{v}^{N+1} \end{bmatrix} = \begin{bmatrix} -\frac{k}{2}\boldsymbol{\Delta v}^N \\ [\frac{\varepsilon^2}{2}\boldsymbol{\Delta} - \frac{1}{2}\mathbb{B}^N]\boldsymbol{u}^N \end{bmatrix} + \boldsymbol{b}^N, \qquad N = 1, 2, \dots\,,$$

where $[\frac{\varepsilon^2}{2}\boldsymbol{\Delta} - \frac{1}{2}\mathbb{B}^N]\boldsymbol{u}^N$ differs from $\boldsymbol{v}^N = [\frac{\varepsilon^2}{2}\boldsymbol{\Delta} - \frac{1}{2}\mathbb{B}^{N-1}]\boldsymbol{u}^N$ in the matrix $\mathbb{B}$ being the current instead of the previous. Defining $\boldsymbol{v}^{N+\frac{1}{2}} := [\frac{\varepsilon^2}{2}\boldsymbol{\Delta} - \frac{1}{2}\mathbb{B}^N]\boldsymbol{u}^N$, and approximating the Laplacian of $\boldsymbol{v}^{N+\frac{1}{2}}$ instead of $\boldsymbol{v}^N$ in the right-hand-side, the previous system could be rewritten as

$$\begin{bmatrix} \mathbb{1} & \frac{k}{2}\Delta \\ -\frac{\varepsilon^2}{2}\Delta + \frac{1}{2}\mathbb{B}^N & \mathbb{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^{N+1} \\ \boldsymbol{v}^{N+1} \end{bmatrix} = \begin{bmatrix} -\frac{k}{2}\boldsymbol{\Delta v}^{N+\frac{1}{2}} \\ \boldsymbol{v}^{N+\frac{1}{2}} \end{bmatrix} + \boldsymbol{b}^N, \qquad N = 1, 2, \dots\,.$$

For the PureMetal problem (2.5), the following semi-discrete equation can be implemented as a semi-implicit time scheme

$$\begin{cases} \dfrac{\psi^{N+1} - \psi^N}{k} &= \dfrac{1}{A(\psi^N)}\Big\{A^2(\psi^N)\Delta\psi^\star + \boldsymbol{C}(\psi^N)\cdot\nabla\boldsymbol{\psi}^\star + \\ &\qquad - [(\psi^N)^2 - 1]\psi^\star - \lambda[(\psi^N)^2 - 1]^2 u^\star\Big\} \\ \dfrac{u^{N+1} - u^N}{k} &= \alpha\Delta u^\star + \dfrac{\psi^{N+1} - \psi^N}{2k} \end{cases}, \qquad \text{on } \Omega\,.$$

where either $\psi^\star = \psi^{N+1}$ and $u^\star = u^{N+1}$, as for the Backward-Euler case, or $\psi^\star = \frac{\psi^{N+1}+\psi^N}{2}$ and $u^\star = \frac{u^{N+1}+u^N}{2}$, as for the Crank-Nicolson scheme. The vector field, $\boldsymbol{C}$, is introduced to simplify the notation

$$C_\zeta(\psi) := \partial_\zeta A^2(\psi) - \sum_{\xi\neq\zeta}\partial_\xi B_{\zeta\xi}(\psi)\,.$$

The corresponding fully-discrete system is then, in the first case,

$$\begin{bmatrix} \mathbb{A}^N - k(\mathbb{A}^N)^2\Delta + k\mathbb{B}^N - k\mathbb{C}^N & k\lambda(\mathbb{B}^N)^2 \\ \frac{1}{2}\mathbb{1} & \mathbb{1} - k\alpha\Delta \end{bmatrix} \begin{bmatrix} \boldsymbol{\psi}^{N+1} \\ \boldsymbol{u}^{N+1} \end{bmatrix} = \begin{bmatrix} \mathbb{A}^N\psi^N \\ \boldsymbol{u}^N - \frac{1}{2}\psi^N \end{bmatrix} + \boldsymbol{b}^N, \qquad N = 1, 2, \dots\,,$$

or, in the second, case

$$\begin{bmatrix} 2\mathbb{A}^N - k(\mathbb{A}^N)^2\Delta + k\mathbb{B}^N - k\mathbb{C}^N & k\lambda(B^N)^2 \\ -\mathbb{1} & 2\mathbb{1} - k\alpha\Delta \end{bmatrix} \begin{bmatrix} \boldsymbol{\psi}^{N+1} \\ \boldsymbol{u}^{N+1} \end{bmatrix} =$$

$$= \begin{bmatrix} 2\mathbb{A}^N + k(\mathbb{A}^N)^2\Delta + k\mathbb{C}^N - k\mathbb{B}^N & -k\lambda(\mathbb{B}^N)^2 \\ -\mathbb{1} & 2\mathbb{1} + k\alpha\Delta \end{bmatrix} \begin{bmatrix} \boldsymbol{\psi}^{N} \\ \boldsymbol{u}^{N} \end{bmatrix} + \boldsymbol{b}^N, \qquad N = 1, 2, \ldots .$$

The matrix blocks $\mathbb{A}^N$, $\mathbb{B}^N$ are time-dependent diagonal matrices defined as

$$\mathbb{A}^N_{ij} := \delta_{ij} A(\psi^N_i) \qquad \mathbb{B}^N_{ij} := \delta_{ij}[(\psi^N_i)^2 - 1], \qquad i, j = 1, 2, \ldots, M ,$$

while $\mathbb{C}^N$ is not diagonal, but can be expressed as a sum of products of diagonal and stencil matrices,

$$\mathbb{C}^N = \sum_\zeta \mathbb{C}^N_\zeta \nabla_\zeta ,$$

where the diagonal matrices are defined as $(\mathbb{C}_\zeta)_{ij} := \delta_{ij} C_\zeta(\psi_i)$ and the matrices ∇_ζ, $(1, \ldots, M-1)$ are the linear discrete operators corresponding to each component of the gradient. These matrices are stencil matrices in the sense that their non-zero pattern is a subset of the pattern of Δ and their coefficients can be accessed, in HYPRE, through the same stencil interface.

For the PhaseField component, therefore, in addition to adding to the semi-structured driver support for solving systems of multiple variables, it is necessary to define a gradient method to the `PhaseField::detail::dw_fd_view` analogously as for the Laplacian operator.

The `PhaseField::Application` base class methods `scheduleInitializeSystemVars` and `schedule-RefineSystemVars` ensure that the tasks for initializing the solver whenever the grid is created or refined is invoked. These methods where introduced earlier to ensure that the `SubProblems` objects are initialized whenever the grid is created or refined, and are scheduled before the application specific steps: `scheduleInitialize` and `scheduleRefine`.

Each application is responsible for creating the variable labels for the stencil and additional matrix entries and for the right-hand-side vector in the DataWharehouse, to update these variable during the simulation and to feed them to the solver. Applications are also responsible to schedule the solver's task for solving the linear system. In [18], details are given about how this is implemented within the `PhaseField::Heat` application.

The implementation for grid with only one level first selects the scheduling method corresponding to the `time_scheme`, invokes it, and then invokes the solver scheduling method for solving the linear system.

More complex is the implementation when AMR grids are used. Firstly, all levels must be scheduled at the same time. Then the implementation splits between the semi-structured driver and the structured diver. As for the one level implementation, both driver's implementations firstly select the scheduling methods corresponding to the `time_scheme` are selected. In any case, two scheduling methods are selected: one for the coarsest level, and one for refined levels.

This distinction is needed as with both driver the task for assembling the linear system on refined level depends on the solution on coarser levels: in the semi-structured case, the previous solution is used to interpolate matrix entries at fine-coarse interfaces; in the structured case, the current solution is used as boundary condition at these interfaces. In the structured branch of the implementation, the next step is to schedule the assemble and solve tasks on the coarsest level and then to loop over all refined levels to schedule the assemble and solve tasks on those levels as well.

The semi-structured branch of the implementation, instead, first loops over all levels to schedule the assemble tasks. In this case, the solver task must be scheduled only once per processor, but before scheduling this task it is necessary to schedule an empty task, to ensure that all pending MPI communications are completed before HYPRE routines are executed. If this task were not scheduled, HYPRE internal communications may overlap with Uintah's own communications causing the program to crash, or to behave unexpectedly.

The scheduling methods introduced here for supporting implicit time schemes are very similar to all other scheduling methods. They specify which variables are required and which are updated by each task.

Thanks to the modifications to the view interfaces described in the previous paragraphs and to the introduction of the `PhasePhield::AdditionalEntries` class, the kernel implementation for these tasks is straightforward. Hereafter, the definition of the kernels used for the Backward Euler time scheme are provided.

```
template<VarType VAR, DimType DIM, StnType STN, bool AMR, bool TST>
void
Heat<VAR, DIM, STN, AMR, TST>::time_advance_solution_backward_euler_assemble_hypre_full
(
  const IntVector & id,
  const FDView < ScalarField<const double>, STN > & u_old,
  View < ScalarField<Stencil7> > & A,
  View < ScalarField<double> > & b
)
{
  std::tuple<Stencil7, double> sys = u_old.laplacian_sys_hypre ( id );

  const Stencil7 & lap_stn = std::get<0> ( sys );
  const double & rhs = std::get<1> ( sys );
  const double a = alpha * delt;

  for ( int i = 0; i < 7; ++i )
    A[id][i] = -a * lap_stn[i];
  A[id].p += 1;
  b[id] = u_old[id] + a * rhs;
}
```

```
template<VarType VAR, DimType DIM, StnType STN, bool AMR, bool TST>
void
Heat<VAR, DIM, STN, AMR, TST>::time_advance_solution_backward_euler_assemble_hypre_rhs
(
  const IntVector & id,
  const FDView < ScalarField<const double>, STN > & u_old,
  View < ScalarField<double> > & b
)
{
  double rhs = u_old.laplacian_rhs_hypre ( id );
  const double a = alpha * delt;

  b[id] = u_old[id] + a * rhs;
}

template<VarType VAR, DimType DIM, StnType STN, bool AMR, bool TST>
void
Heat<VAR, DIM, STN, AMR, TST>::time_advance_solution_backward_euler_assemble_hypresstruct_full
(
  const IntVector & id,
  const FDView < ScalarField<const double>, STN > & u_old,
  View < ScalarField<Stencil7> > & A_stencil,
  HypreSStruct::AdditionalEntries * A_additional,
  View < ScalarField<double> > & b
)
{
  std::tuple<Stencil7, HypreSStruct::AdditionalEntries, double> sys = u_old.
       laplacian_sys_hypresstruct ( id );

  const Stencil7 & lap_stn = std::get<0> ( sys );
  HypreSStruct::AdditionalEntries & lap_add = std::get<1> ( sys );
  const double & rhs = std::get<2> ( sys );
  const double a = alpha * delt;

  for ( int i = 0; i < 7; ++i )
    A_stencil[id][i] = -a * lap_stn[i];
  A_stencil[id].p += 1;
  for ( auto & entry : lap_add )
    *A_additional += -a * entry;
  b[id] = u_old[id] + a * rhs;
}
```

```
template<VarType VAR, DimType DIM, StnType STN, bool AMR, bool TST>
void
Heat<VAR, DIM, STN, AMR, TST>::time_advance_solution_backward_euler_assemble_hypresstruct_rhs
(
  const IntVector & id,
  const FDView < ScalarField<const double>, STN > & u_old,
  View < ScalarField<double> > & b
)
{
  double rhs = u_old.laplacian_rhs_hypresstruct ( id );
  const double a = alpha * delt;

  b[id] = u_old[id] + a * rhs;
}
```

From the code above, it is evident how views and the additional entries structures implemented in the PhaseField component achieve the target of allowing the application developer to easily implement new applications for the simulation of phase field problems approximated with finite-differences. The same kernel can be used on each cell regardless of its position on the grid and of the geometry around it. Few remarks are to be made. First, all seven entries of the Stencil7 class are always used regardless of the problem dimension (*17*, *56*), as this is the common practice in Uintah. Moreover, the syntax for updating additional entries (*59*–*60*) is almost identical to the one for updating stencil entries (*56*–*57*).

Similar implementations have been added also to the Benchmark02 application.

8.3 Validation

To assess the reliability of the implementation of implicit time stepping methods here introduced a set of simulations is performed to analyse the convergence to the analytical solution of the Heat problem with Dirichlet boundary conditions as both the space and time discretization steps are reduced.

Since the explicit analytical expression of the solution to the heat problem (2.25) with homogeneous Dirichlet conditions and initial conditions (5.1) over the domain $\Omega = [-L, L]^d$ know, as described in §5.2, it is possible to evaluate at every timestep a global discrete error of the solution (5.7) and of its first and second order derivatives as (5.11). These errors are therefore computed for different values of the grid step, h, and time step, k, and the speed with which these errors are converging to zero is compared to the order of the truncation error that is computed a priori for different maximum number of AMR levels.

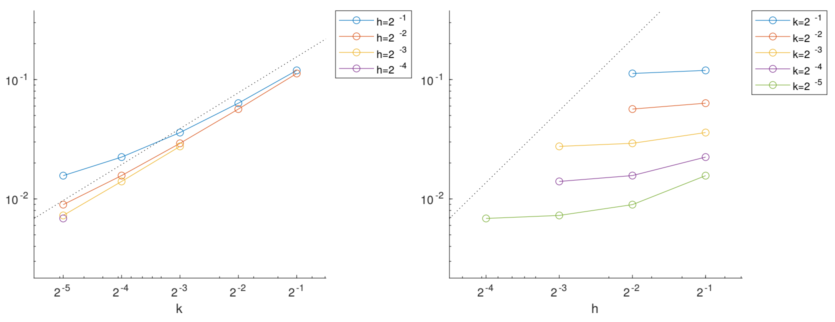
Using the leading-order error formulae for finite differences approximations (2.27) as in §6.3 it can be shown that the truncation error is linear in time and quadratic in space for the Backward–Euler (BE) scheme and quadratic both in time and space for the Crank–Nicolson

(CN) method when the discretisation grid is uniform.

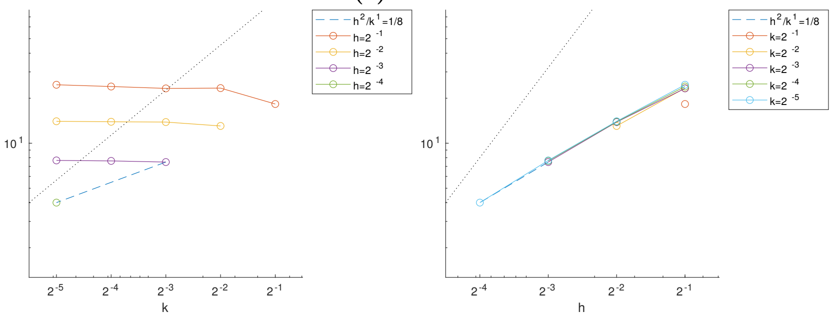
The values for the spatial and temporal discretization steps considered are $h = 2^{-i}$ with $(i = 1, 2, \dots, 5)$ and $k = 2^{-j}$ with $(j = 1, 2, \dots, 5)$. Both implicit time stepping schemes are unconditionally stable.

Tests have been performed with most of the available structured and semi-structured solvers, and all gave very similar results. In Figures 8.1–8.3 the discrete 1-norm, $\sum_N |\, \mathbf{err}_p^N \,|$, of the computed global discrete errors (5.11) are plotted in logarithmic scale, first, against the timestep and, then, against the grid step for the Backward–Euler time scheme using the structured LGM-RES linear solver. Values of error corresponding to the same choice of the other discretisation parameter are joint with solid lines, while values with the same $\frac{h^2}{k}$ ratio are joint with dashed lines.

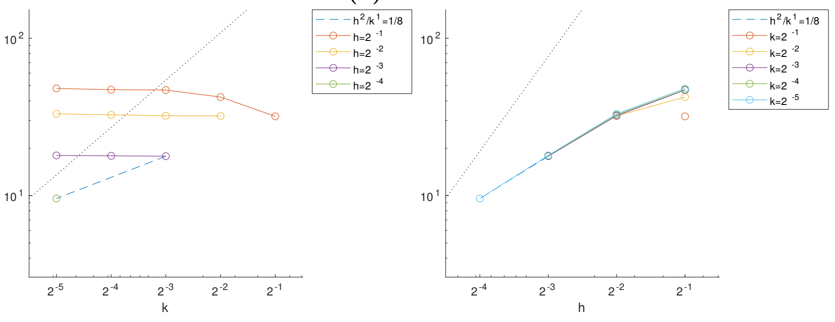
In Figures 8.4–8.6 analogue plots are provided for the Crank–Nicolson scheme using the structured GMRES linear solver. The dashed lines, in this case, join cases with the same $\frac{h^2}{k}$ ratio.



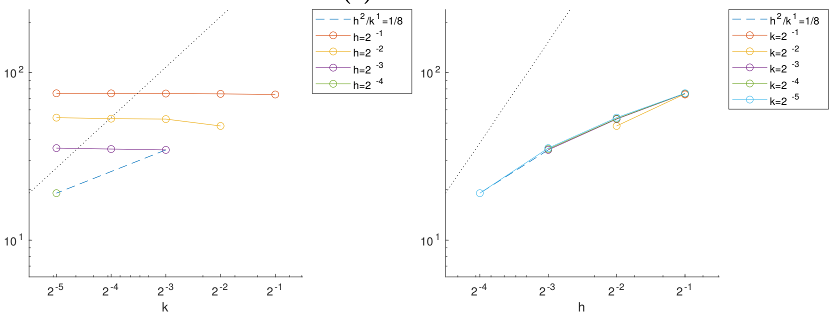
(a) one level



(b) two levels

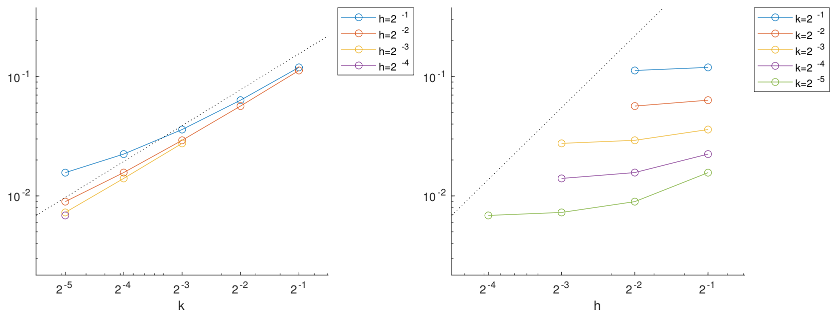


(c) three levels

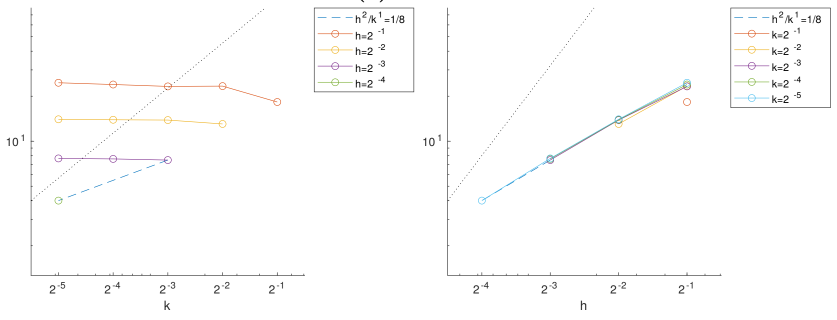


(d) four levels

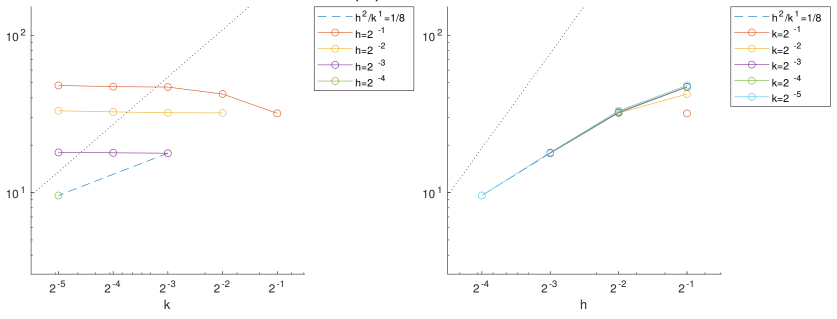
Figure 8.1: Convergence analysis of the global L^2 error for the Heat problem using Backward-Euler time stepping and the structured LGMRES linear solver. Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (left) and on the grid size (right) for different numbers of AMR levels. Graphs proportional to k^1 and h^2 are plotted for reference as a dotted lines.



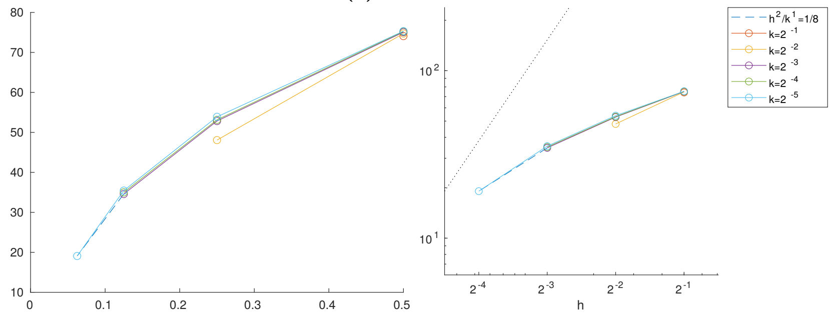
(a) one level



(b) two levels

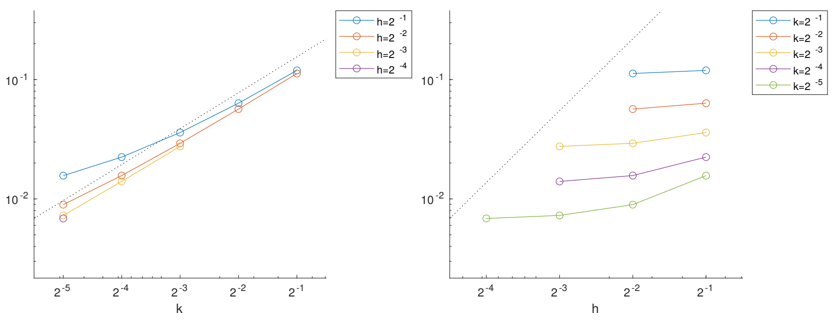


(c) three levels

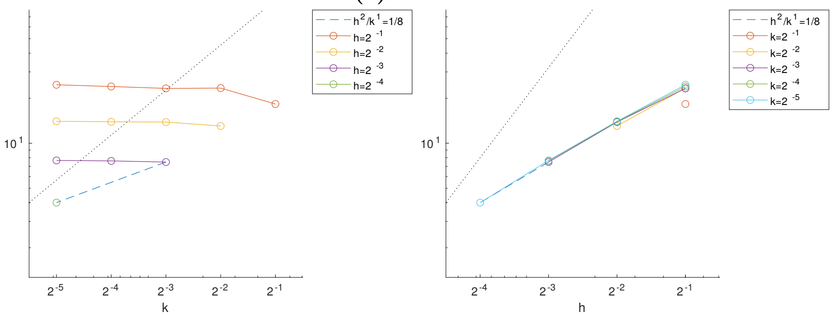


(d) four levels

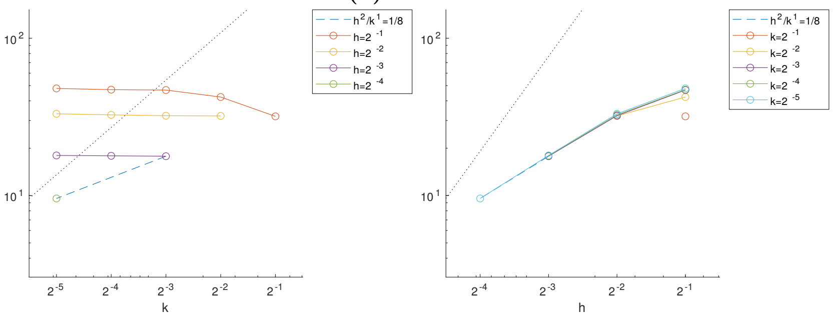
Figure 8.2: Convergence analysis of the global H^1_0 error for the Heat problem using Backward-Euler time stepping and the structured LGMRES linear solver. Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (left) and on the grid size (right) for different numbers of AMR levels. Graphs proportional to k^1 and h^2 are plotted for reference as a dotted lines.



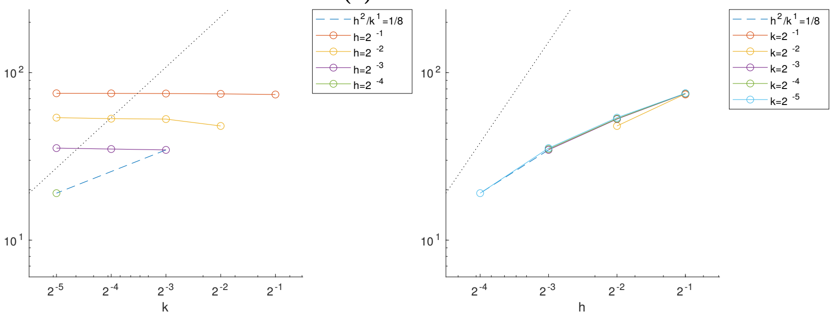
(a) one level



(b) two levels

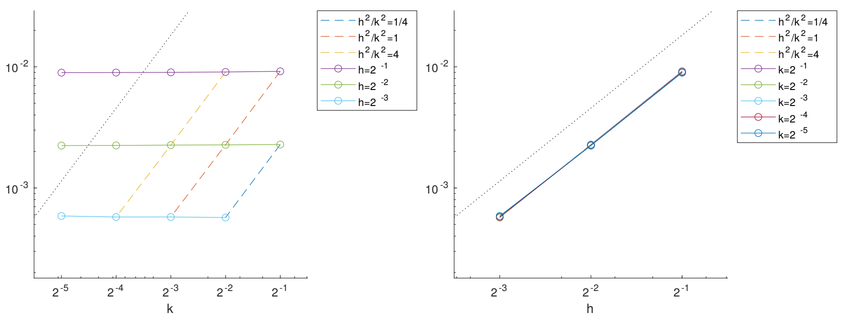


(c) three levels

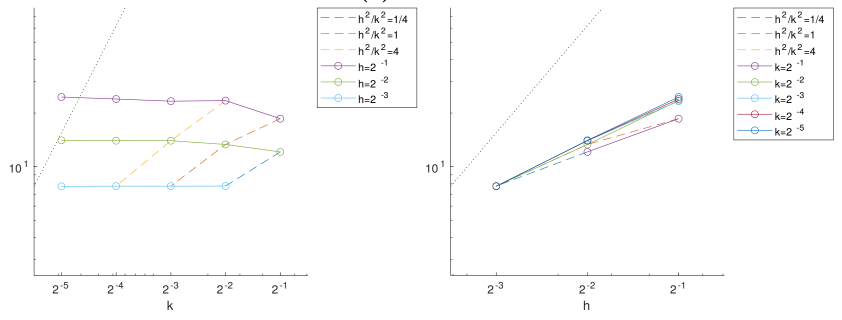


(d) four levels

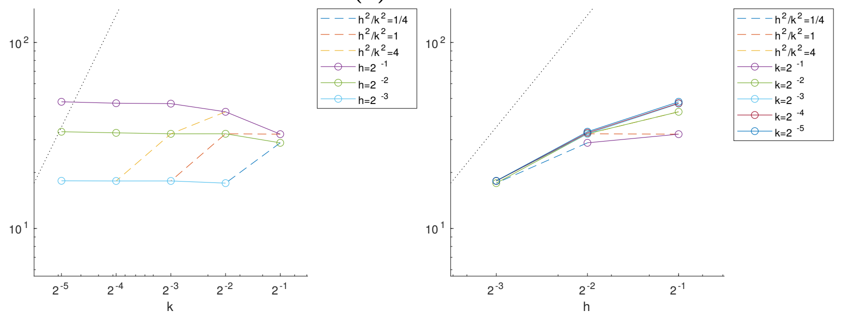
Figure 8.3: Convergence analysis of the global H^2_0 error for the Heat problem using Backward-Euler time stepping and the structured LGMRES linear solver. Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (left) and on the grid size (right) for different numbers of AMR levels. Graphs proportional to k^1 and h^2 are plotted for reference as a dotted lines.



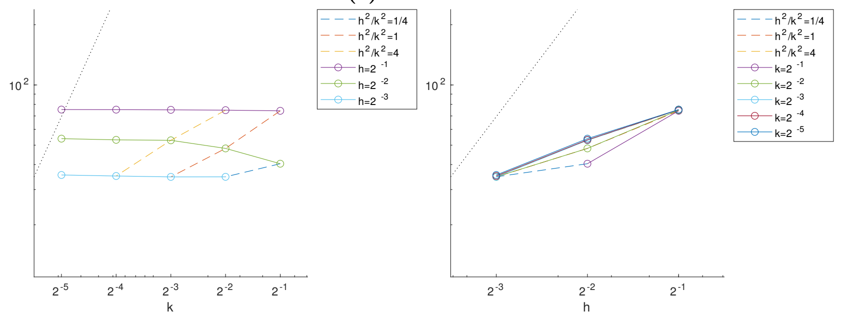
(a) one level



(b) two levels

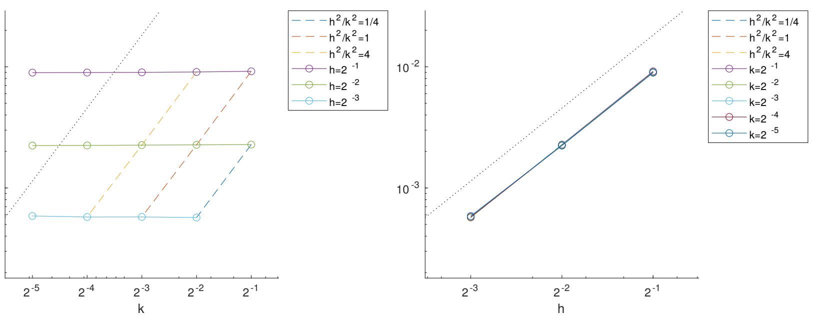


(c) three levels

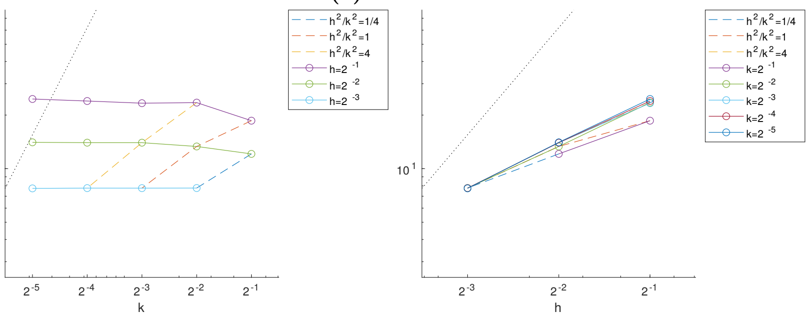


(d) four levels

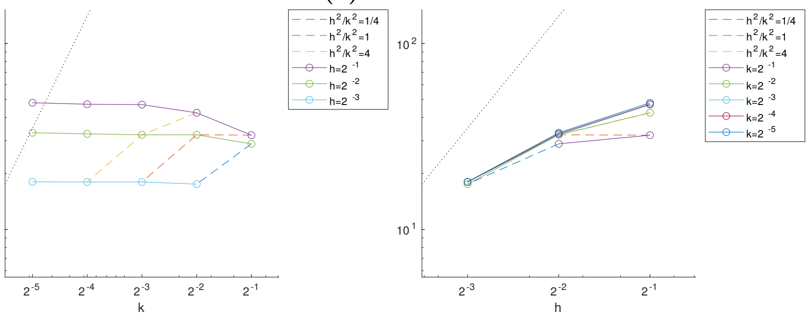
Figure 8.4: Convergence analysis of the global L^2 error for the Heat problem using Crank-Nicholson time stepping and the structured GMRES linear solver. Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (left) and on the grid size (right) for different numbers of AMR levels. Graphs proportional to k^2 and h^2 are plotted for reference as a dotted lines.



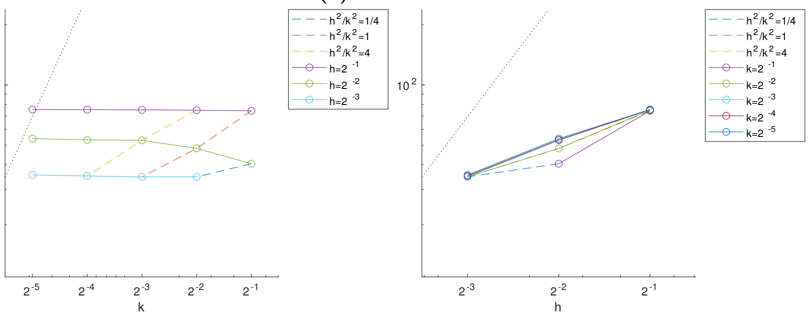
(a) one level



(b) two levels

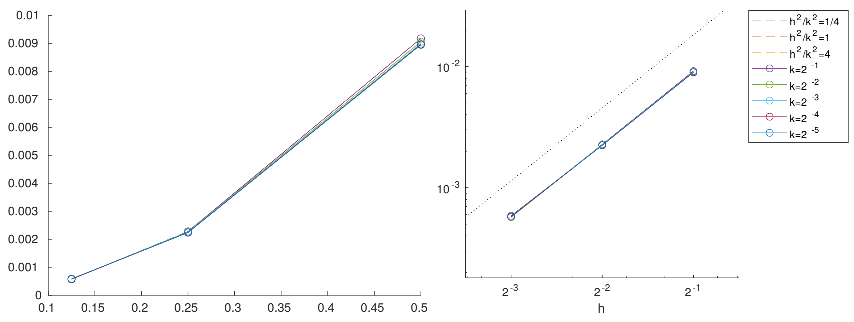


(c) three levels

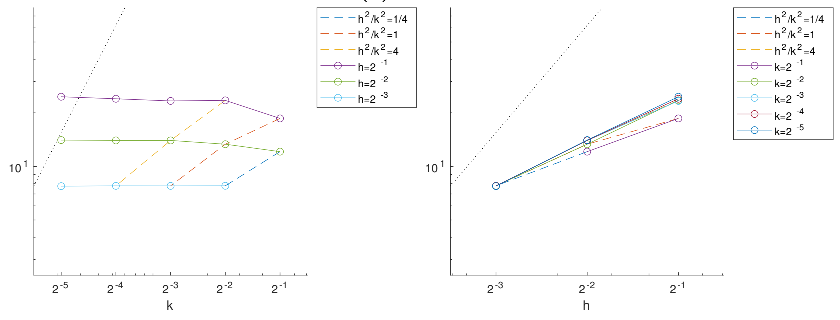


(d) four levels

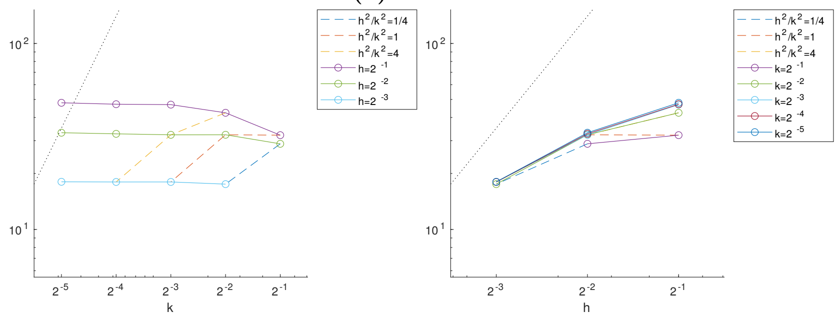
Figure 8.5: Convergence analysis of the global H_0^1 error for the Heat problem using Crank-Nicholson time stepping and the structured GMRES linear solver. Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (left) and on the grid size (right) for different numbers of AMR levels. Graphs proportional to k^2 and h^2 are plotted for reference as a dotted lines.



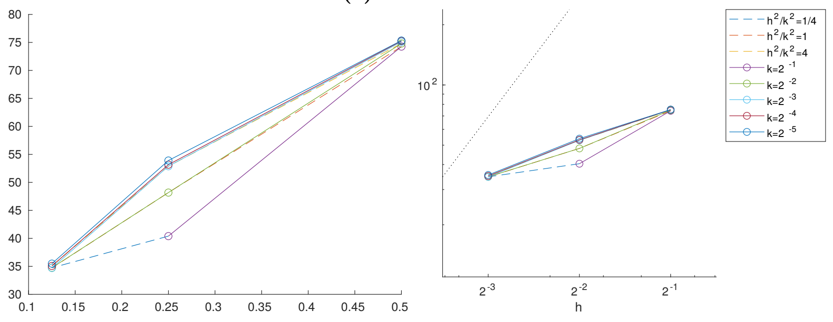
(a) one level



(b) two levels



(c) three levels



(d) four levels

Figure 8.6: Convergence analysis of the global H_0^2 error for the Heat problem using Crank-Nicholson time stepping and the structured GMRES linear solver. Its 1-norm over time for different discretization parameters is shown as dependent on the timestep (left) and on the grid size (right) for different numbers of AMR levels. Graphs proportional to k^2 and h^2 are plotted for reference as a dotted lines.

Looking at the solid line in the plots on the left-hand side of Figure 8.1, it is evident how halving the timestep is not beneficial with AMR grids, since the truncation error in this case is dominated by the spatial component. Looking at the graphics on the right-hand side, it can be noticed that an order of convergence in space is lost when adaptivity is introduced regardless of the number of levels. This means that the interpolation error for the Heat application contributes to the overall error more than the truncation component. Also the convergence order in time is affected by the introduction of adaptive levels, as evident from looking at the dashed lines.

For the Crank–Nicolson scheme, the spatial component of the error dominates the temporal in all the considered cases. Theoretical convergence orders are however observed only without AMR. When adaptive grid are used, an order of convergence is lost with respect to both the spatial and temporal discretisation steps.

Chapter 9

Parallel Performance

The parallel scalability results that are reported in this chapter are all performed on homogeneous architecture using only CPU cores. However, since computer architectures such as those designed for exascale computations are rapidly evolving, Uintah developers are continuously working on implementing solutions that are portable across heterogeneous platforms while exploiting all the computational power provided by these architectures.

In [85], for example, the author introduces a new unified, portable, Single Instruction Multiple Data (SIMD) primitive which allows both intrinsics-based vectorization on CPUs and many-core architectures, as well as Single Instruction Multiple Threads (SIMD) based execution on GPUs. Thanks to this unified primitive, together with the Kokkos framework, it is possible to develop explicitly vectorized code which is also portable across heterogeneous system.

Kokkos [12], as other Performance Portability Layers (PPL), promise portability across different platforms thanks to their ability to interact with different underlying programming models (e.g., CUDA, HIP, OpenMP, etc) through a unified higher level interface. They hide the low-level details of the underlying programming models and simplify the task of efficiently run the same code on heterogeneous nodes. Alternative to Kokkos, are OCCA [66], RAJA [43], and SYCL [54]. In [42], the authors show performance improvements up to 4.4x when using a heterogeneous MPI+Kokkos task scheduler in the Uintah Computational Framework.

Recently [41], significant work has been done, also, in combining Uintah and Hedgehog with the end goal of building an asynchronous many-task runtime system specializing in both node-level and large-scale performance while increasing their accessibility with portable abstractions. Hedgehog is a general-purpose performance-oriented C++17 headers-only library specializing in maximizing single-node utilization with emphasis on heterogeneous nodes. Lying at the other extreme is Uintah which emphasizes on large-scale simulations performance on major HPC systems.

Uintah developers also actively worked on optimizing the HYPRE solver for manycore and

GPU architectures [86, 84].

It has to be noted that, despite the PhaseField component have not been tested on heterogeneous platforms during this research, that all implementations of the PhaseField component and of its application have been developed using the unified portable interfaces provided by Uintah, and that, therefore, the PhaseField component may benefit from the aforementioned active developments of the Uintah Computational Framework toward exascale computing.

9.1 Scalability Test Setup

To analyse the properties of the parallel implementation within Uintah a series of simulations has been performed using an increasing number of computational nodes. All simulations have been performed on ARC4, the HPC service of the University of Leeds. Its standard nodes have 40 cores and 192GB of memory each and an SSD within the node with 170GB. All system CPUs are Intel Xeon Gold 6138 CPUs ('Sky Lake').

Both weak and strong scalability tests have been considered. *Weak scalability* tests measure how a software performs when the number of cores is increased while keeping the working load constant, i.e. the problem size per processor, is kept constant. Weak scalability tests are a useful tool, for example, in understanding the parallel performance of a software when its resources are fully exploited.

Conversely, *strong scalability* tests are used to measure the performance of a software when the number of cores is increased while keeping the global problem size constant, thus decreasing the processors' load. These tests are a useful tool to quantify what fraction of the whole calculation is parallelised: in fact, only if the computation is fully parallelised, an inverse proportionality is observed between computation time and number of processors.

Three sets of tests have been performed to assess the performance of the newly implemented PhaseField component of the Uintah Computational Framework. In §9.2, the performance of the solver is tested with the PureMetal application on both two and three dimension geometries, while increasing the number of adaptive levels. The Heat application is then used to assess the parallel scalability of the component while using structured linear solvers (§9.3) and semi-structured (§9.4) from HYPRE, while increasing the number of adaptive levels. For all simulations only cell-centred discretisation have been used.

9.2 AMR Scalability: Assessment of Performance with Increasing Level of Adaptivity

The first family of parallel performance test performed on the PhaseField component is intended to assess the scalability of the implementation as the number of AMR level is increased from one to eight, which is the maximum allowed by Uintah.

These tests focuses on the scalability of the explicit solver on those timesteps that does not perform any additional tasks other than those necessary to update the solution. To avoid taking into account the initial timestep and any regrid timestep, fixed geometry configurations have been used. The result obtained using fixed semi structured geometries is still representative of runs where the mesh is adaptively updated throughout the simulation since error estimation tasks are still scheduled and executed; therefore the same tasks that would have been performed on a truly adaptive grid are completed.

To analyse the solver parallel performance in two dimensions, three different grid arrangements have been considered:

Case I : each additional level refines the left half of the previous most refined level;

Case II : each additional level refines the bottom-left quart of the previous most refined level;

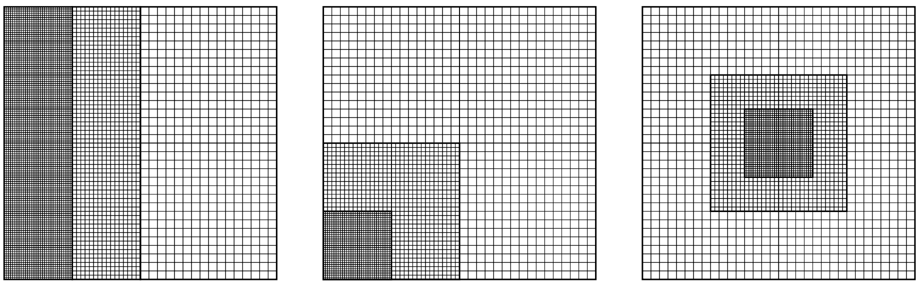
Case III : each additional level refines the central quart of the previous most refined level.

As an example of these arrangements, in Figure 9.1 a representation of the three cases is given for three AMR levels. In Case I, each level adds twice as many cells as in the previous most refined level, in the other Cases, each level has the same number of cells. In all simulation the patch size is fixed to 16×16 cells, with the coarsest level having four patches for the smallest load case. To consider problems of increasing size, different geometries are considered by iteratively quadrupling the number of patches on each level. In this way weak scalability can be assessed comparing simulations with the same *load*, that is with the same ratio of cells per core.

The different configurations have been chosen to compare cases where different proportions of cells on fine/coarse interfaces with respect to the overall number of cells. When more interface cells are present, more MPI communication is performed. The three Cases allows to infer how these may affect the scalability performance of the PhaseField component when AMR is enabled.

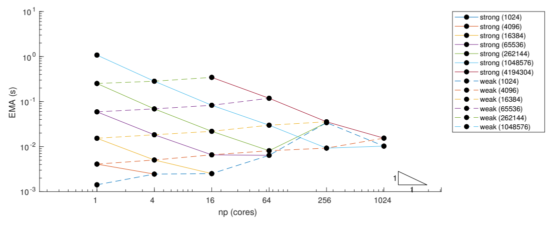
In Figures 9.2–9.4 the execution time (EMA) of the simulations performed is reported as a function on the number of cores (np) utilized for the different cases respectively.

It can be observed that, in all three cases, strong scalability is almost optimal when AMR is used. Only when one AMR level is used, it can be observe that for the simulations with the

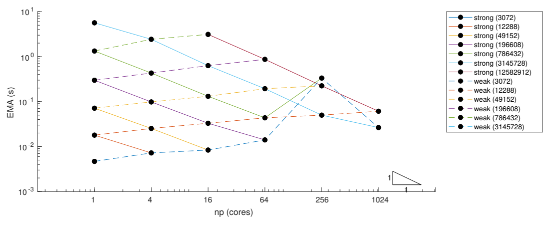


(a) Case I **(b)** Case II **(c)** Case III

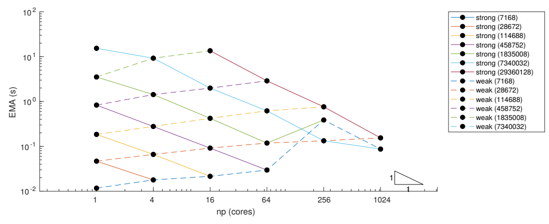
Figure 9.1: Refinement configurations with three AMR levels in 2D.



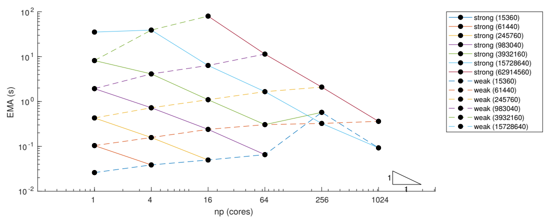
(a) one level



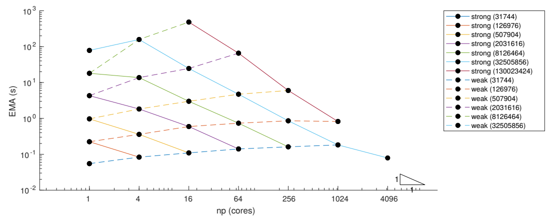
(b) two levels



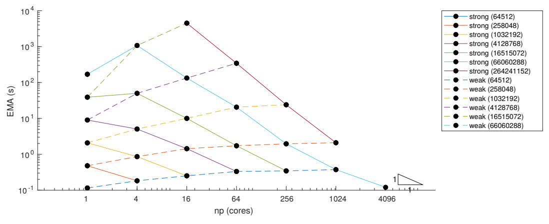
(c) three levels



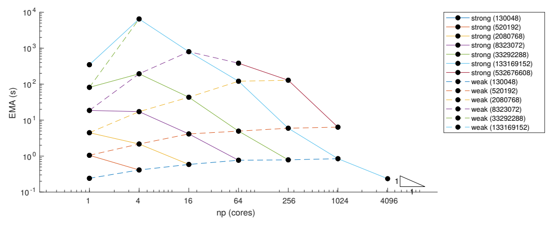
(d) four levels



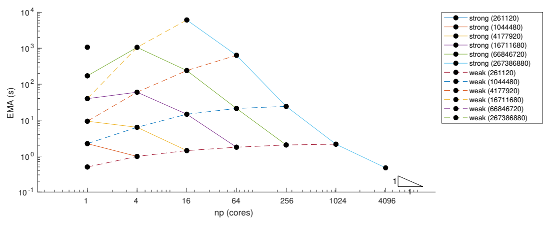
(e) five levels



(f) six levels

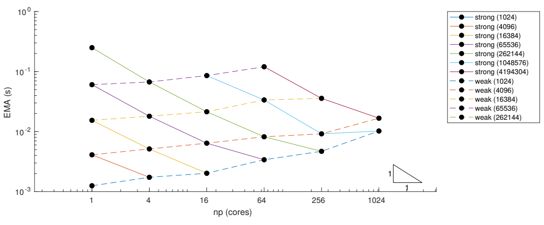


(g) seven levels

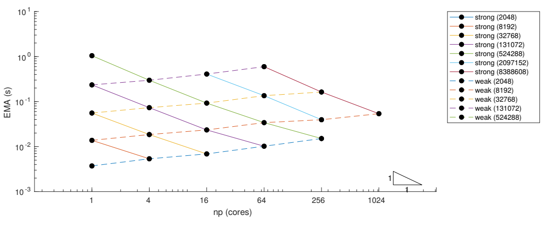


(h) eight levels

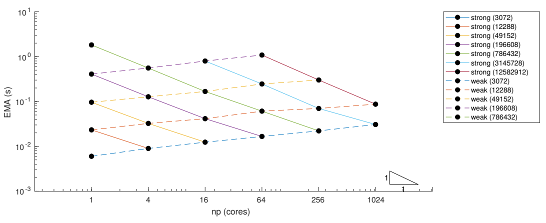
Figure 9.2: Scalability for the 2D PureMetal application with AMR, Case I. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



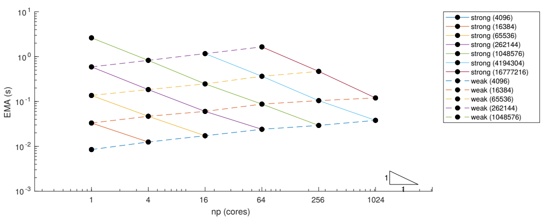
(a) one level



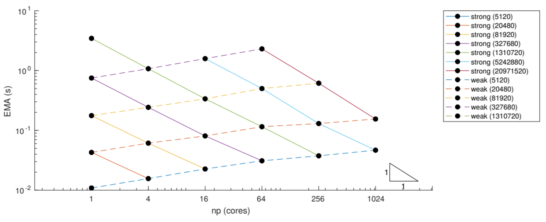
(b) two levels



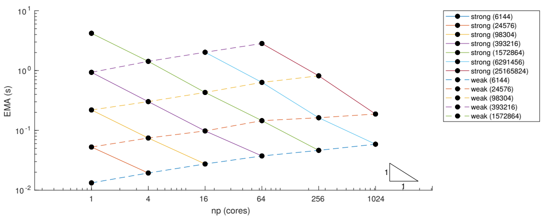
(c) three levels



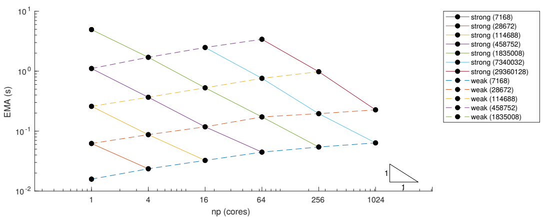
(d) four levels



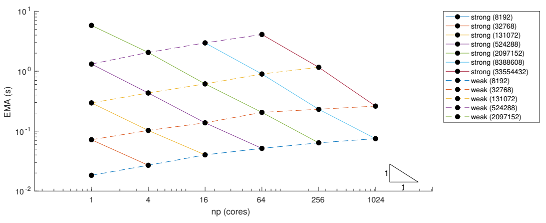
(e) five levels



(f) six levels

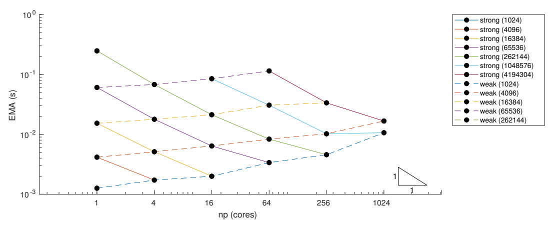


(g) seven levels

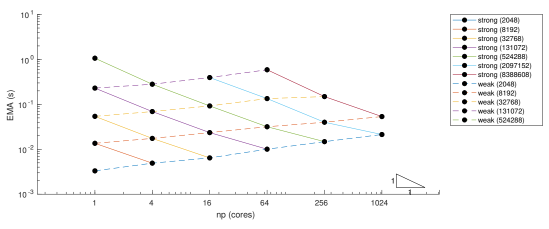


(h) eight levels

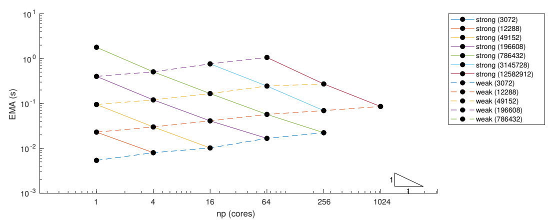
Figure 9.3: Scalability for the 2D PureMetal application with AMR, Case II. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



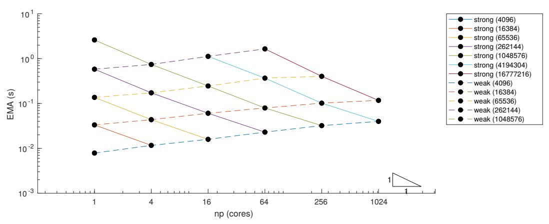
(a) one level



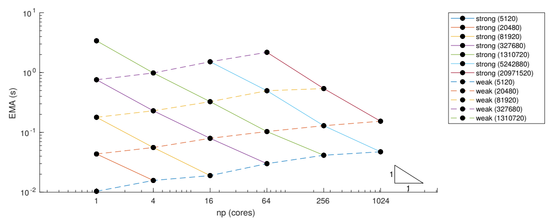
(b) two levels



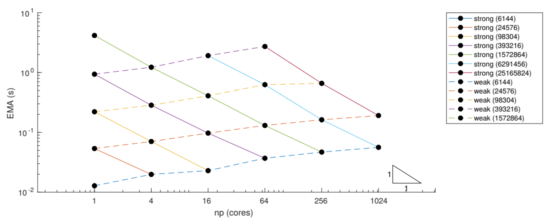
(c) three levels



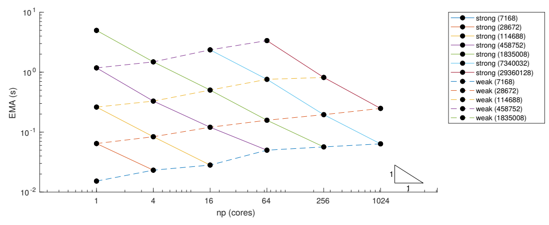
(d) four levels



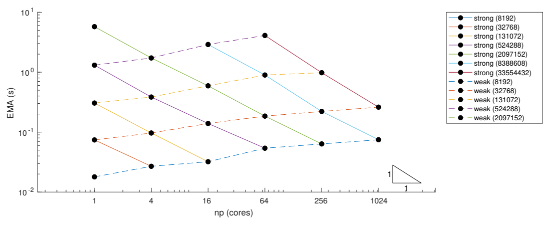
(e) five levels



(f) six levels



(g) seven levels



(h) eight levels

Figure 9.4: Scalability for the 2D PureMetal application with AMR, Case III. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.

larger geometries strong scalability is less then optimal.

The weak scalability paths, moreover, show that for smaller loads and fewer levels the communication overhead introduced by increasing the number of cores can dominate the speed increase in the computation itself. There is a lower limit on the number of cells per processor that must be taken into account when considering to increase the number of cores for a simulation.

Another observation is that the slope of the weak scalability paths is steeper as the load increases. This behaviour is more evident for tests using more AMR levels.

In the other cases (II and III), no dependency of the slope of the weak scalability paths on the load is appreciable. The main difference between Case I and Case II and III, is the ratio of the number of cells between adjacent levels. In Case I, the size of the FCI interfaces is smaller than in the others. This behaviour, therefore, could be explained by the fact that the number of levels and cells per level is increased faster than the number of cores.

Also for the tests in three dimensions, three different grid arrangements have been considered:

Case I: each additional level refines the left half of the previous most refined level;

Case II: each additional level refines the front-bottom-left octave of the previous most refined level;

Case III: each additional level refines the central octave of the previous most refined level.

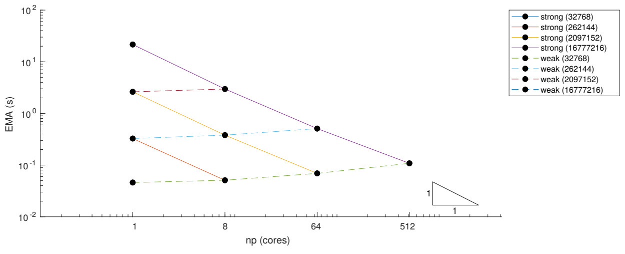
In all simulation the patch size is fixed to $16 \times 16 \times 16$ cells, with the coarsest level having eight patches for the smallest load case.

To consider problems of increasing size, different geometries are considered by iteratively multiplying by eight the number of patches on each level. In this way weak scalability can be assessed comparing simulations with the same *load*, considering the sequence $8^0, 8^1, \ldots$ for the number of cores.

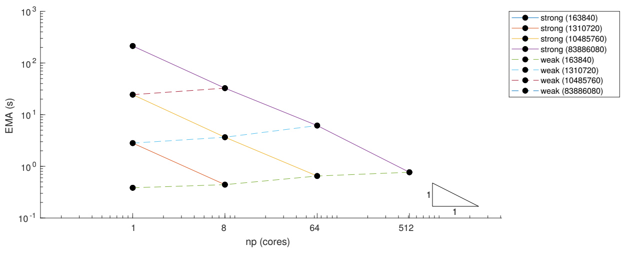
In Figures 9.5–9.7 the execution time (EMA) of the simulations performed is reported as a function on the number of cores (np) utilized for the different cases respectively.

Fewer combinations of loads/number of cores have been tested, but for all three AMR configurations, the strong scalability observed is almost ideal. Also the slope of the weak scalability paths is minimal, which is very promising. The execution time of each time step in 3D is quite expensive in the range of processors considered and that limited the amount of tests performed in this research. However, the tests performed suggest that increasing the number of cores will produce an optimal increase in the computation speed.

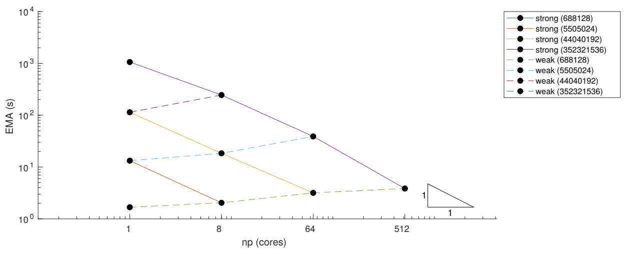
The sizes of the patches considered in the tests above is relatively small. Increasing the size of the patches may be beneficial in some applications since it reduces the amount of communication between nodes processors owning adjacent patches since the ratio between communication and load is proportional to the ratio between the patch perimeter and area. Such tests have not



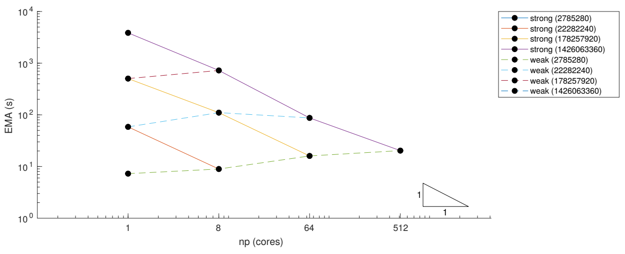
(a) one level



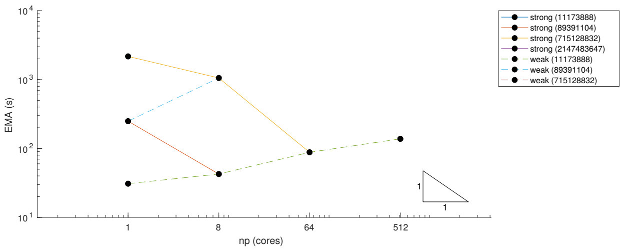
(b) two levels



(c) three levels

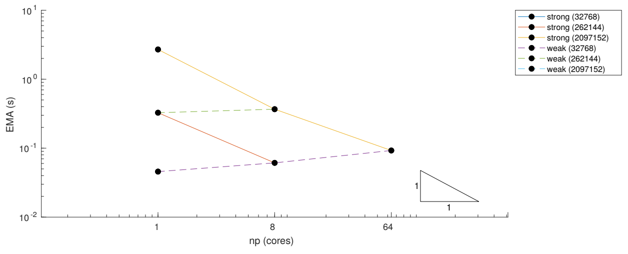


(d) four levels

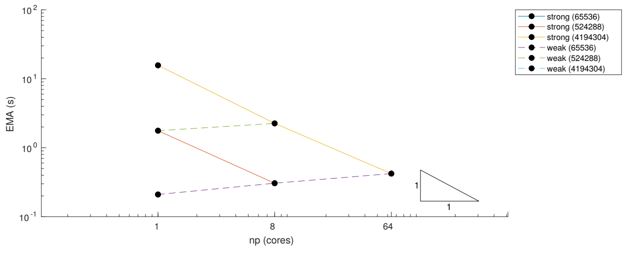


(e) five levels

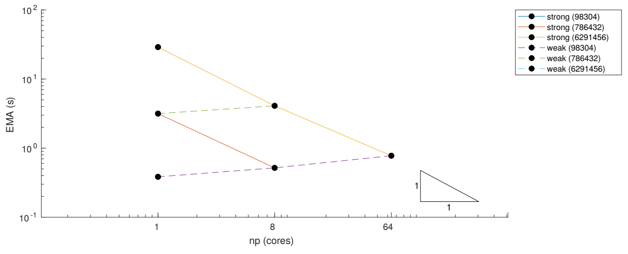
Figure 9.5: Scalability for the 3D PureMetal application with AMR, Case I. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



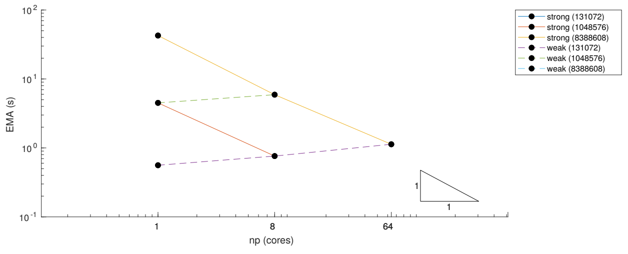
(a) one level



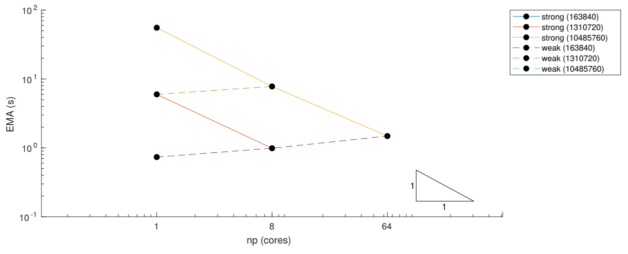
(b) two levels



(c) three levels

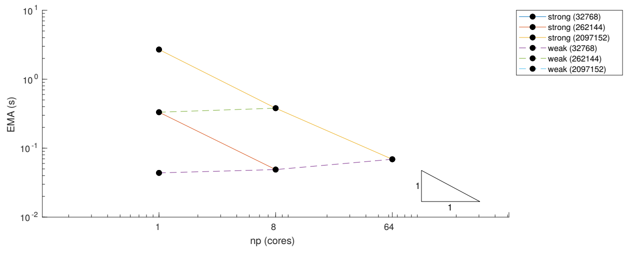


(d) four levels

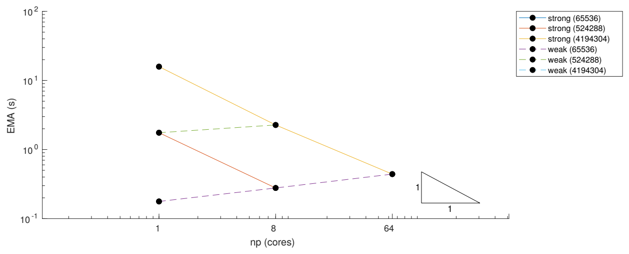


(e) five levels

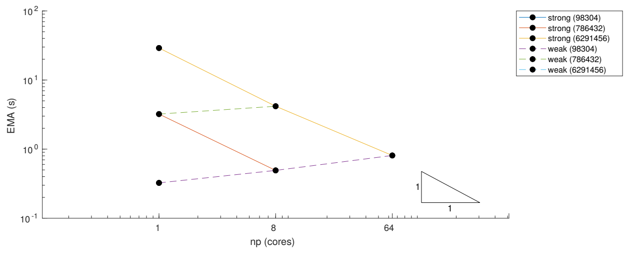
Figure 9.6: Scalability for the 3D PureMetal application with AMR, Case II. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



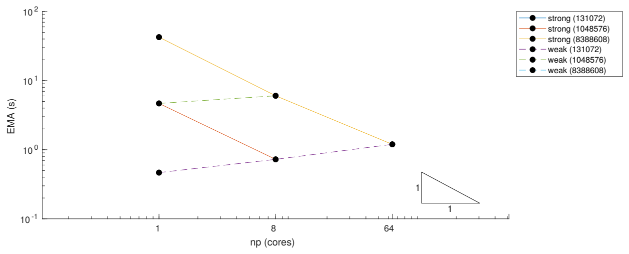
(a) one level



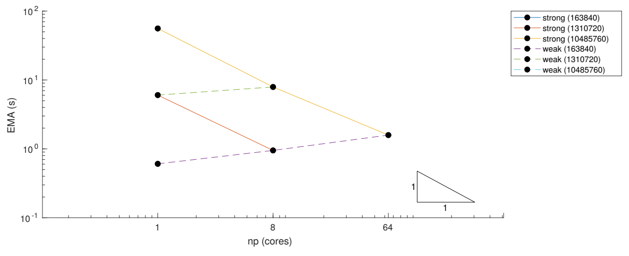
(b) two levels



(c) three levels



(d) four levels



(e) five levels

Figure 9.7: Scalability for the 3D PureMetal application with AMR, Case III. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.

been performed during this research work and may be object of future work. However, using large patches may be unfeasible in those applications, such as some PhaseField applications, where only a thin layer of refined cells is required to model a sharp interface.

9.3 AMR Scalability: Assessment of Performance with Structured Linear Solvers

For assessing the behaviour of the PhaseField component in combination with the structured linear solver provided by HYPRE, the same three AMR configurations used in the previous section are considered for each of the suitable structured linear solver from the HYPRE library against the Heat application in two dimensions. They have been used with the default choice of parameters and unpreconditioned.

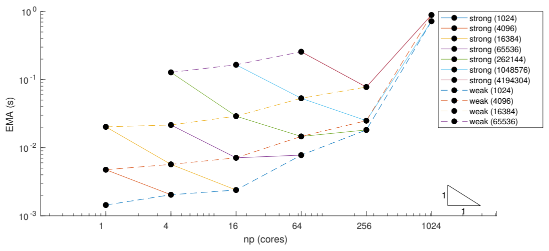
The execution times (EMA) observed with different solvers show similar scalability behaviour. For this reason only plots for tests sake performed using the FlexGMRES structured solver are reported here (Figures 9.8–9.10). Analogue plots for tests using the other structured solvers are available online [19].

In all three AMR grid configurations, the tests performed using the Heat application revealed that the strong scalability is almost ideal until 4^4 cores with all the structured solvers.

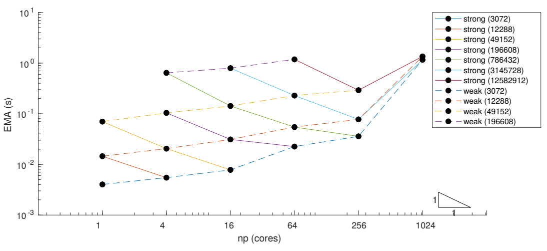
The same Heat tests revealed that the GMRES, LGMRES, and the FlexGMRES weak scalability is worse with smaller problems – with fewer levels or smaller loads – if compared with more computationally challenging problems. This behaviour –which is even more evident when the with the BiCGStab and CycRED solvers– is most probably due to the processors being assigned not enough computational work to compensate the increasing computational overhead due to the handing an increasing number of parallel tasks across the available cores.

The multigrid SMG solver suffers more heavily than the others from this minimum load limitation. For more than 4^3 cores the execution times per timestep are almost the same independently of the problem size, specifically on the smaller grids with less than seven levels.

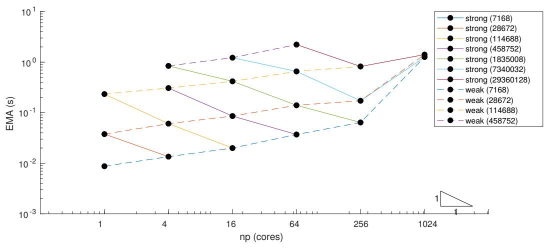
The behaviour of the other multigrid solver, PFMG, is less regular and it is more difficult to identify patterns and infer parallel behaviours, especially in Case I. For example, with seven and eight AMR levels, strong scalability patterns can be observed for different grid sizes, but with four levels, increasing the number of cores does not appreciably result in speeding up the timestep execution.



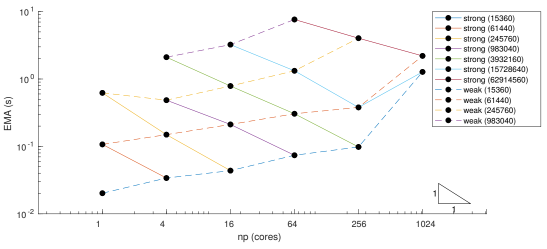
(a) one level



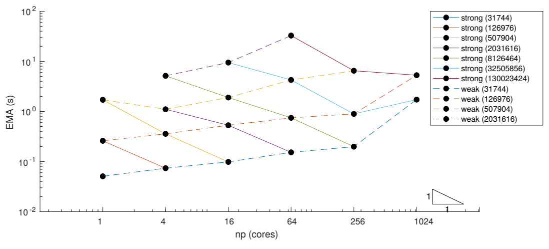
(b) two levels



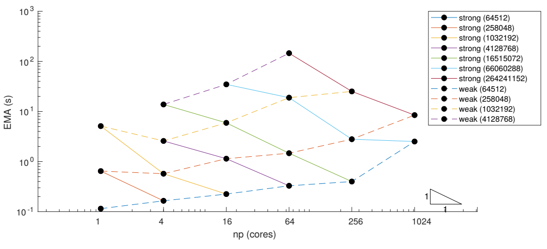
(c) three levels



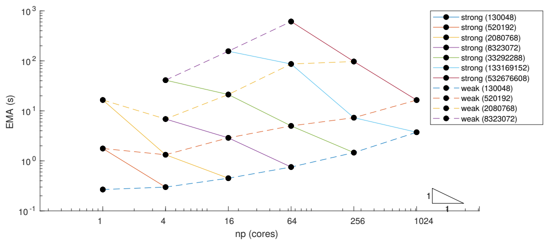
(d) four levels



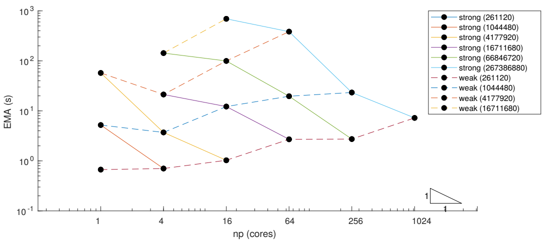
(e) five levels



(f) six levels

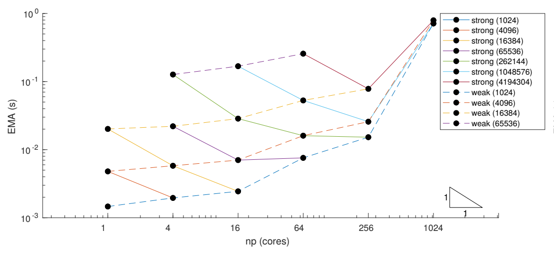


(g) seven levels

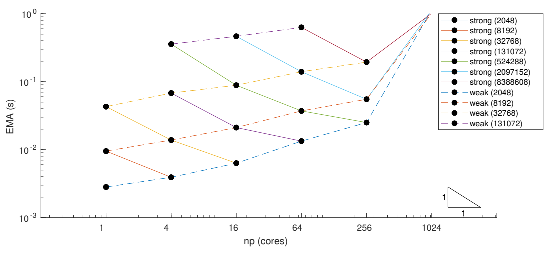


(h) eight levels

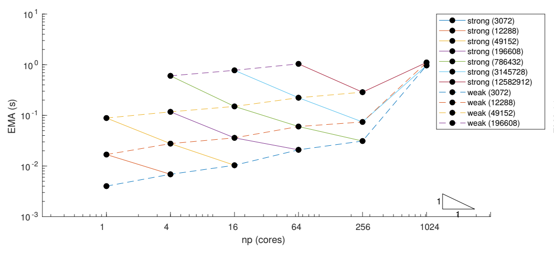
Figure 9.8: Scalability for the 2D Heat application with the FlexGMRES structured linear solver, Case I. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



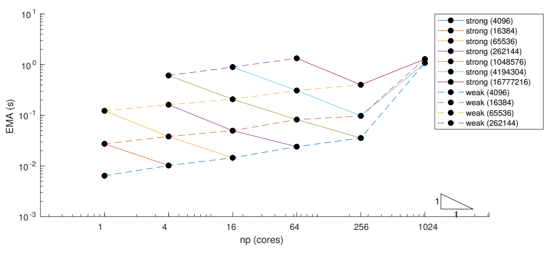
(a) one level



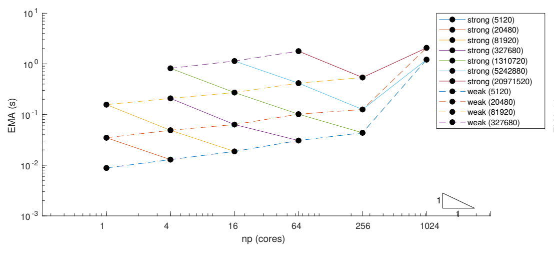
(b) two levels



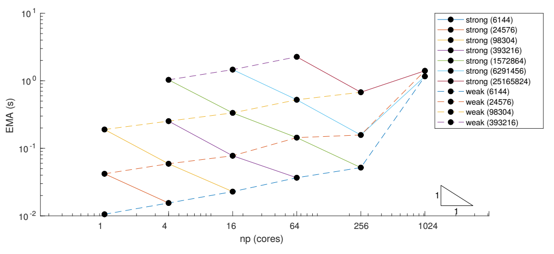
(c) three levels



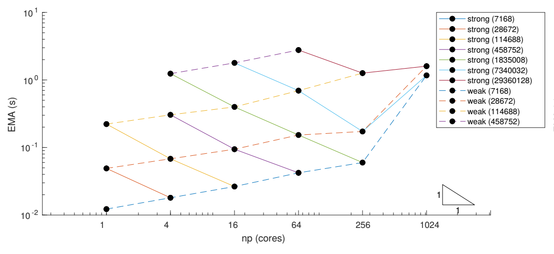
(d) four levels



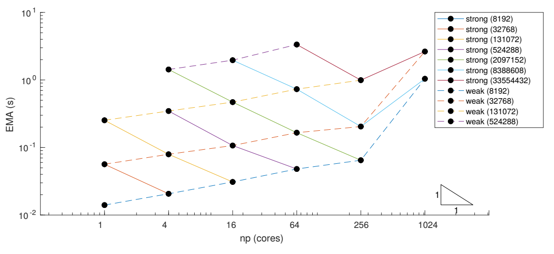
(e) five levels



(f) six levels

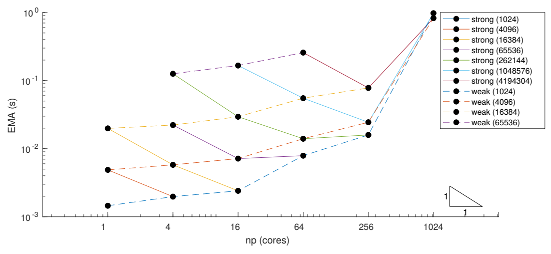


(g) seven levels

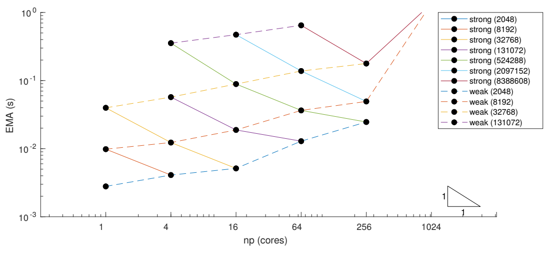


(h) eight levels

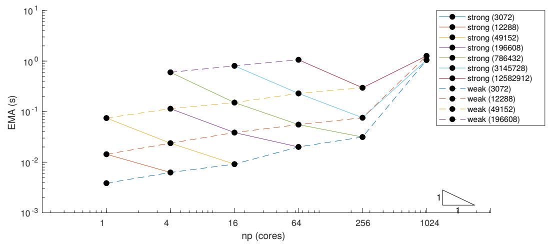
Figure 9.9: Scalability for the 2D Heat application with the FlexGMRES structured linear solver, Case II. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



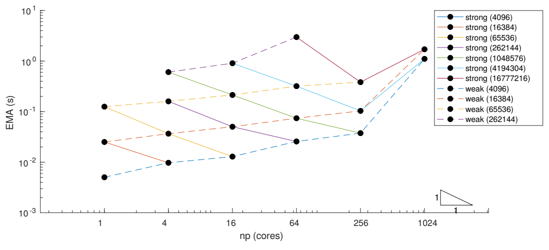
(a) one level



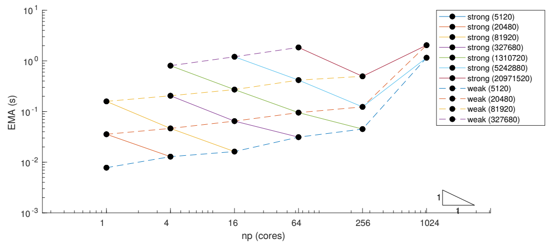
(b) two levels



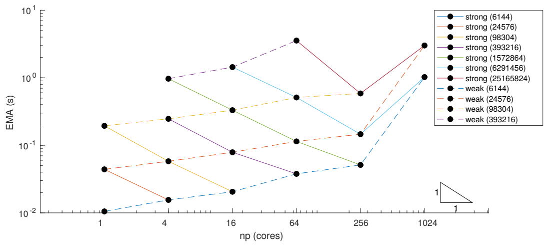
(c) three levels



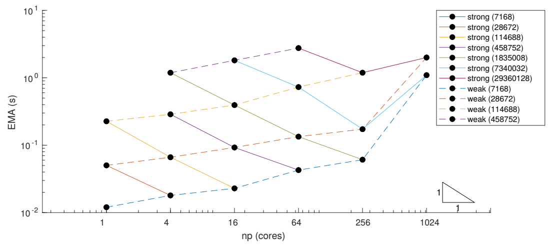
(d) four levels



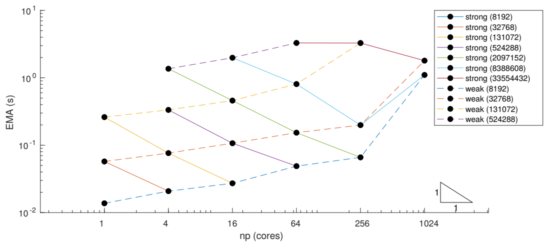
(e) five levels



(f) six levels



(g) seven levels



(h) eight levels

Figure 9.10: Scalability for the 2D Heat application with the FlexGMRES structured linear solver, Case III. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.

9.4 AMR Scalability: Assessment of Performance with Semi-Structured Linear Solvers

Fewer tests have been run for assessing the parallel behaviour of the PhaseField component in combination with the semi-structured linear solver provided by HYPRE. As for the previous sections, the same three AMR configurations are considered for each of the suitable semi-structured linear solver from the HYPRE library for the 2D Heat problem. All the solvers have been used with the default choice of parameters and unpreconditioned.

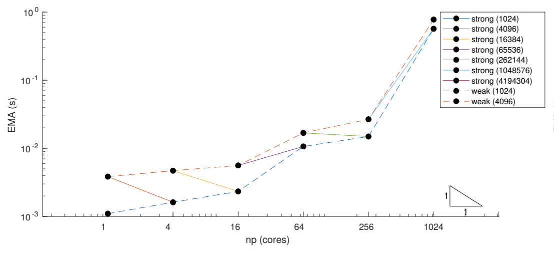
Figures 9.11– 9.13 show the scalability patterns for the FlexGMRES semi-structured solver. Similar plots can be accessed online [19] for the other semi-structured solvers. From these plots, it is evident that optimal scalability is achieved by the PhaseField solver only when enough load is given to each processor. The same scalability patterns are showed also by the GMRES and the LGMRES semi-structured solvers.

In the case of the BiCGStab semi-structured solver, the impact of MPI communication on the execution times seems to be less evident.

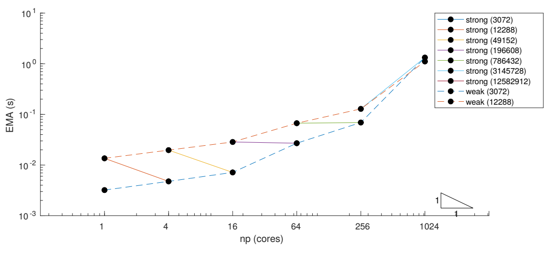
The semi-structured multigrid solver present some issues with MPI communication within HYPRE overlapping with those between Uintah tasks. During this research a lot of effort has been put trying to fix these issues both on the Uintah's side and within HYPRE, but it was not possible to make the solver run smoothly with more than few MPI processes. For these reason no scalability tests are presented here.

A limited number of tests run successfully with the domain decomposition solver, Split. For this solver, the few strong scalability paths are almost ideal. Weak scalability paths for the same load are consistent across the different combinations of number of AMR levels and geometry configurations.

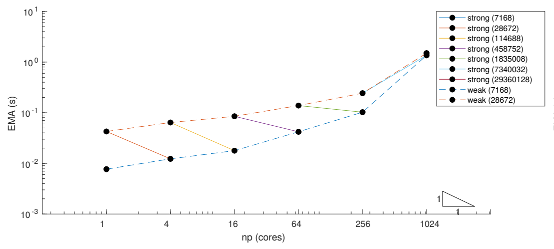
In general it seems that, when HYPRE solvers are used, almost ideal scalability is achieved only when the problem size is very large that the task computations take longer than MPI communication. This limit was not evident from the explicit results in §9.2. An explanation of this could be that the optimal patch size for Uintah produces boxes that are too small for the HYPRE implementation. This aspect should be investigated further in a future research which should take into account the recent progresses in [86, 84].



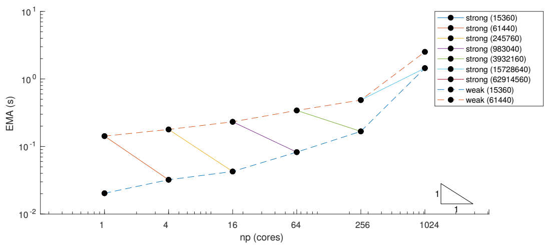
(a) one level



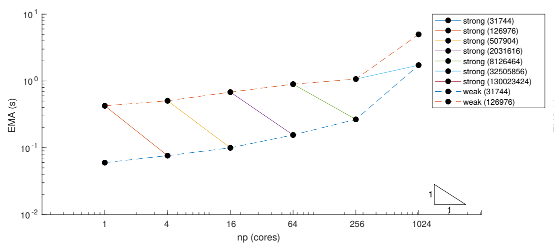
(b) two levels



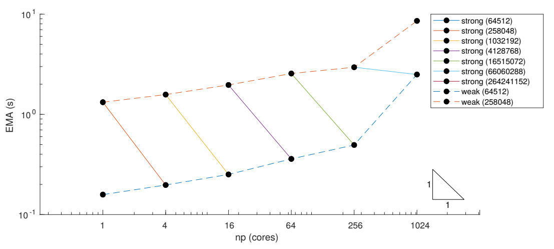
(c) three levels



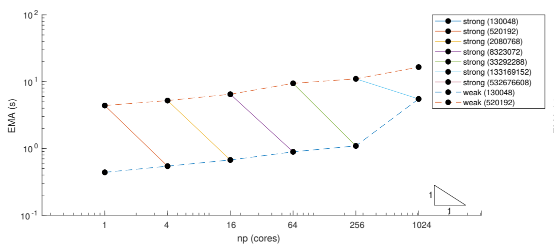
(d) four levels



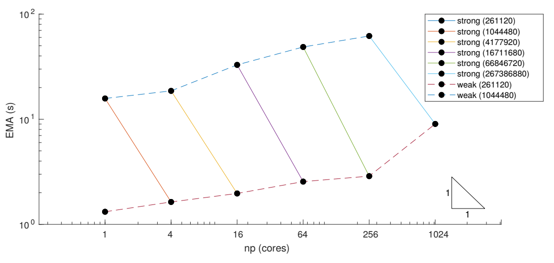
(e) five levels



(f) six levels

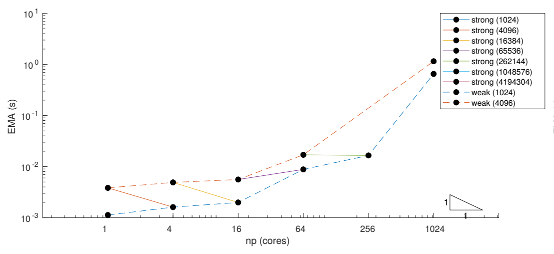


(g) seven levels

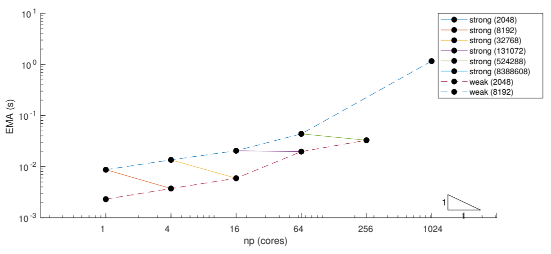


(h) eight levels

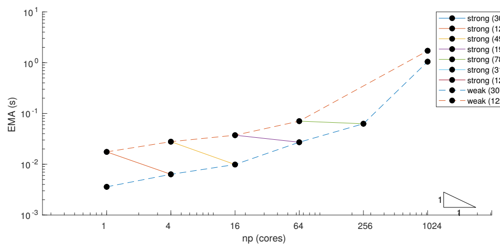
Figure 9.11: Scalability for the 2D Heat application with the FlexGMRES semi-structured linear solver, Case I. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



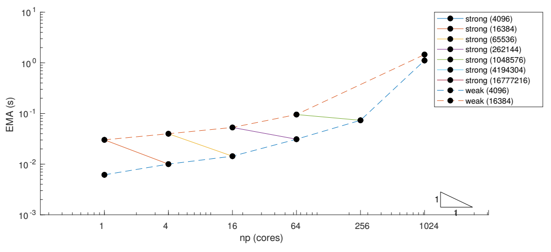
(a) one level



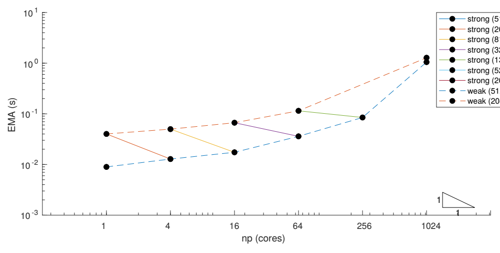
(b) two levels



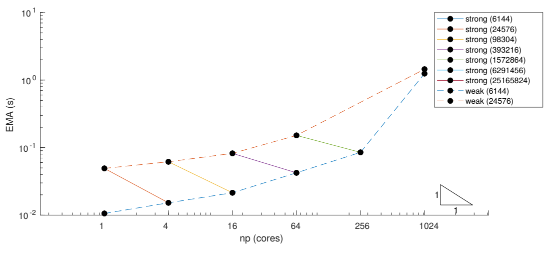
(c) three levels



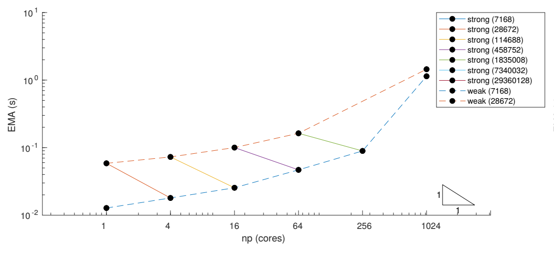
(d) four levels



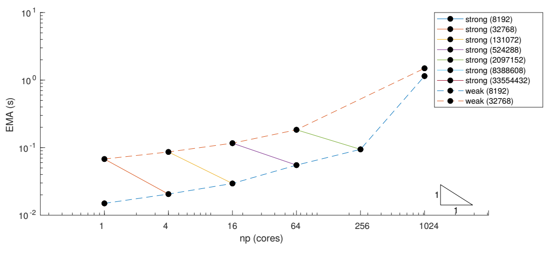
(e) five levels



(f) six levels

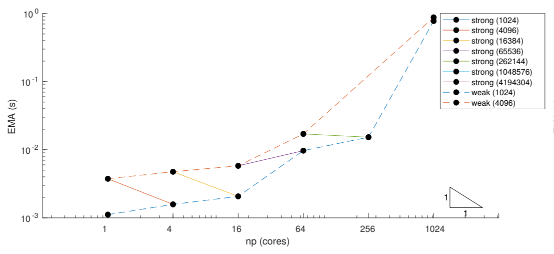


(g) seven levels

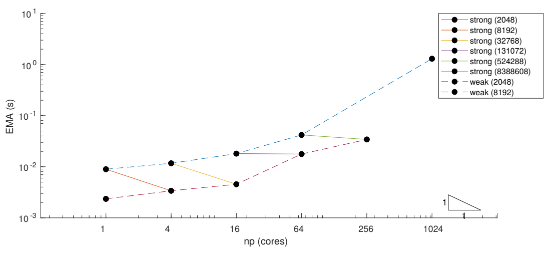


(h) eight levels

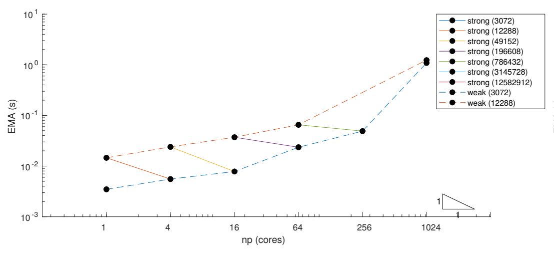
Figure 9.12: Scalability for the 2D Heat application with the FlexGMRES semi-structured linear solver, Case II. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.



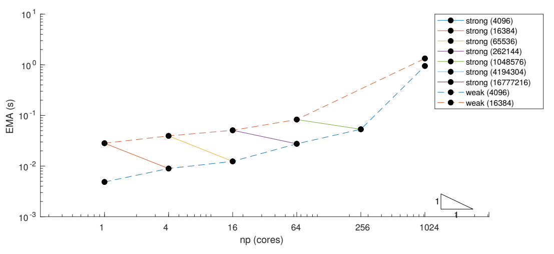
(a) one level



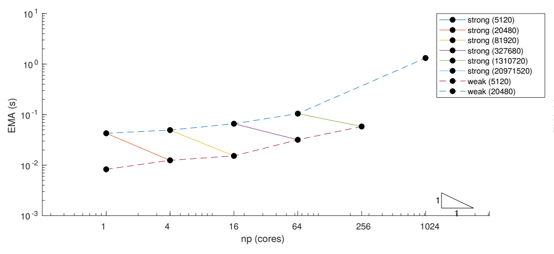
(b) two levels



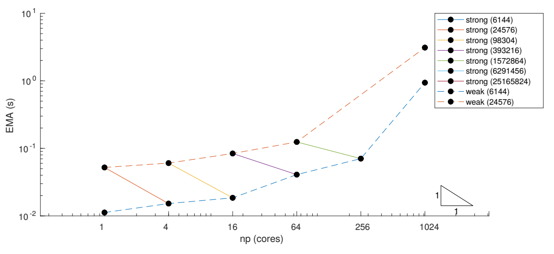
(c) three levels



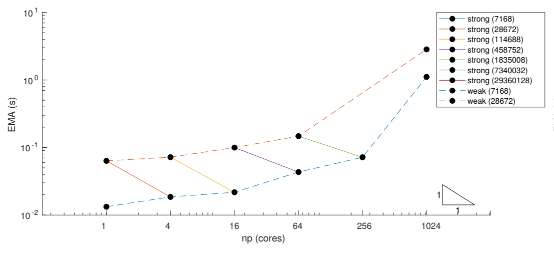
(d) four levels



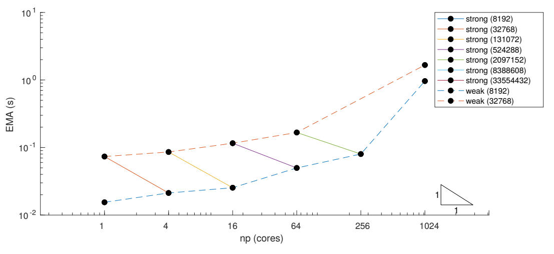
(e) five levels



(f) six levels



(g) seven levels



(h) eight levels

Figure 9.13: Scalability for the 2D Heat application with the FlexGMRES semi-structured linear solver, Case III. Week scalability paths are drawn with dashed lines. Strong scalability paths (solid) can be compared to the reference triangle corresponding to ideal strong scalability.

Chapter 10

Conclusions and Further Developments

The main result of this research work has been the implementation of the PhaseField component within the Uintah Computation Framework. This new software tool allows the implementation of highly scalable parallel solvers for complex system of partial differential equations more conveniently than before. The UCF, in fact, already provided a convenient framework that effectively separated the low-level complexity associated to managing efficiently thousands of processes working in parallel from the high-level implementation of a numerical solver for a specific application. However, before this research, each application had to be developed independently as an individual component. Each of these components used their own numerical techniques and, as a result, techniques implemented for a specific application not necessarily worked for another application.

In the new PhaseField component, effort has been made to ensure that these techniques could have been used together: for example, in Uintah, some application was already using adaptive mesh refinement and some other were implementing implicit time stepping, but these techniques could not be used together effectively.

Another achievement of this research work is the implementation of an abstract interface to the finite differences discretisation of differential operators. This additional abstraction serves three purposes: first, it allows the application developer to implement the finite differences discretisation of any system of PDEs without the need to focus on edge cases such as boundary conditions and fine/coarse interfaces; second, it avoids any code duplication making the component code easier to maintain; and, third, it provides an efficient mechanism to switch between general and edge cases implementations.

Novelty

No other structured adaptive grid software offers the same level of abstraction and flexibility that this new component offers. The framework developed in the PhaseField component is at the same time portable across multiple platform, highly optimized while offering a high level interface that effectively decouples the definition of the discretized model from the implementation of the solver. The syntax offered by the view interfaces has been demonstrated to provide the application developer a way to implement new finite differences/element solvers for general applications that can benefit from SAMR and implicit time stepping over large system with unprecedented ease. All low level complexity is effectively hidden from the application developer: parallelism and adaptivity are hidden in Uintah's framework, boundary and fine/coarse interface conditions and the assembly of implicit matrices are hidden in PhaseField's view framework, while the linear solver integration is hidden in the new SStructSolver component.

Summary of Contributions

The list of contributions from this research can be grouped in software and application contributions.

Software contributions

- abstract interface for fields independent of their discretisation (cell centred or vertex based)
- abstract interface for discrete differential operators independent on the stencil and problem dimension
- boundary and fine/coarse interface conditions automatically imposed according to runtime specifications
- linear system assembly with abstract interface which hides any complexity due to AMR geometry and boundary conditions
- immediate access to all HYPRE solvers and preconditioner
- automatic conversion between Uintah and HYPRE structures

Applications contributions

- template implementation of explicit and implicit solvers for the heat problem as reference implementation for PhaseField applications.
- implementation of Allen-Chan and Cahn-Hillard solvers as benchmark applications
- implementation of the PureMetal solver which showcases a typical phase field problem application.

Future Work

In the future, the PhaseField component development may follow two main paths. The first one consists in the implementation of more complex models, such as multi phase field models, which can exploit fully the solving capability of the component. The second path consists in the improvement of the PhaseField component and SStructSover implementations themselves. For example, since template class are used extensively throughout the new finite differences framework, the compilation time of the new component could be improved by adding appropriate explicit class instantiations.

Further improvements to the component implementation could be aiming to simplify further the interface for the application developer, which at the moment must use different methods for approximating differential operators depending on which time-stepping method and solver is used.

Also the process for specifying task dependencies could be simplified. In particular inter level dependencies when using AMR could be handled automatically by the PhaseField component.

Overlapping the two paths could be the implementation of additional time stepping methods. At present, each time scheme must be implemented for each application since the responsibility of their implementation is of the application developer. This is because task dependencies for each timestep are specified in the application implementation. However, it may be convenient to implement an abstract interface to time stepping methods similar to the finite differences interface here implemented.

Bibliography

[1] Samuel M. Allen and John W. Cahn. "A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening". In: *Acta Metallurgica* 27.6 (1979), pp. 1085–1095. ISSN: 0001-6160. DOI: 10.1016/0001-6160(79)90196-2.

[2] W. E. Arnoldi. "The principle of minimized iterations in the solution of the matrix eigenvalue problem". In: *Quarterly of Applied Mathematics* 9 (1951), pp. 17–29. DOI: 10.1090/qam/42792.

[3] Steven F. Ashby and Robert D. Falgout. "A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations". In: *Nuclear Science and Engineering* 124.1 (1996), pp. 145–159. DOI: 10.13182/NSE96-A24230.

[4] A. H. Baker, E. R. Jessup, and T. Manteuffel. "A Technique for Accelerating the Convergence of Restarted GMRES". In: *SIAM Journal on Matrix Analysis and Applications* 26.4 (2005), pp. 962–984. DOI: 10.1137/S0895479803422014.

[5] Peter W. Bates and Paul C. Fife. "Spectral comparison principles for the Cahn-Hilliard and phase-field equations, and time scales for coarsening". In: *Physica D: Nonlinear Phenomena* 43.2 (1990), pp. 335–348. ISSN: 0167-2789. DOI: 10.1016/0167-2789(90)90141-B.

[6] Martin Berzins et al. "Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices". In: *SIAM Journal on Scientific Computing* 38.5 (2016), S101–S122. DOI: 10.1137/15M1023270.

[7] P. C. Bollada et al. "Three dimensional thermal-solute phase field simulation of binary alloy solidification". In: *Journal of Computational Physics* 287 (2015), pp. 130–150. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2015.01.040.

[8] J. Bosch, C. Kahle, and M. Stoll. "Preconditioning of a Coupled Cahn-Hilliard Navier-Stokes system". In: *Communications in Computational Physics* 23 (2018), pp. 603–628. DOI: 10.4208/cicp.OA-2017-0037.

[9] P. N. Brown, Robert D. Falgout, and Jim E. Jones. "Semicoarsening multigrid on distributed memory machines". In: *SIAM Journal on Scientific Computing* 21.5 (2000), pp. 1823–1834.

[10] A. D. Brydon et al. "Simulation of the densification of real open-celled foam microstructures ". In: *Journal of the Mechanics and Physics of Solids* 53.12 (2005), pp. 2638–2660. ISSN: 0022-5096. DOI: 10.1016/j.jmps.2005.07.007.

[11] John W. Cahn and John E. Hilliard. "Free Energy of a Nonuniform System. I. Interfacial Free Energy". In: *The Journal of Chemical Physics* 28.2 (1958), pp. 258–267. DOI: 10.1063/1.1744102.

[12] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[13] B. Chalmers. *Principles of Solidifications*. New York, NY: John Wiley & Sons, 1964.

[14] J. G. Charney, R. Fjörtoft, and J. Von Neumann. "Numerical Integration of the Barotropic Vorticity Equation". In: *Tellus* 2.4 (1950), pp. 237–254. DOI: 10.3402/tellusa.v2i4.8607.

[15] C. C. Chen, Y. L. Tsai, and C. W. Lan. "Adaptive phase field simulation of dendritic crystal growth in a forced flow: 2D vs 3D morphologies". In: *International Journal of Heat and Mass Transfer* 52.5 (2009), pp. 1158–1166. ISSN: 0017-9310. DOI: 10.1016/j.ijheatmasstransfer.2008.09.014.

[16] Long-Qing Chen. "Phase-Field Models for Microstructure Evolution". In: *Annual Review of Materials Research* 32.1 (2002), pp. 113–140. DOI: 10.1146/annurev.matsci.32.112001.132041.

[17] Andrew Christlieb et al. "High accuracy solutions to energy gradient flows from material science models". In: *Journal of Computational Physics* 257 (2014), pp. 193–215. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2013.09.049.

[18] Jon Matteo Church. *PhaseField Component Manual*. 2022. URL: https://github.com/jonmatteochurch/uintah/blob/phase-field/doc/Components/PhaseField/manual.pdf.

[19] Jon Matteo Church. *Uintah Phase Field Scalability Tests Repository*. 2022. URL: https://github.com/jonmatteochurch/uintah-phasefield_scalability.

[20] Jon Matteo Church et al. "High Accuracy Benchmark Problems for Allen-Cahn and Cahn-Hilliard Dynamics". In: *Communications in Computational Physics* 26 (2019), pp. 947–972.

[21] P. Colella et al. *Chombo Software Package for AMR Applications Design Document*. Tech. rep. Lawrence Berkely National Laboratory, Applied Numerical Algorithms Group, Computational Research Division, Dec. 2009. URL: https://crd.lbl.gov/assets/pubs_presos/chomboDesign.pdf.

[22] Phillip Colella et al. “A Cartesian grid embedded boundary method for hyperbolic conservation laws”. In: *Journal of Computational Physics* 211.1 (2006), pp. 347–366. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2005.05.026.

[23] A. R. Conn, N. I. M. Gould, and P. L. Toint. *Trust Region Methods*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2000. ISBN: 978-0-898-71460-9.

[24] James O. Coplien. “Curiously Recurring Template Patterns”. In: *C++ Report* 7.2 (Feb. 1995), pp. 24–27. ISSN: 1040-6042.

[25] Jean-Marc Debierre et al. “Phase-field approach for faceted solidification”. In: *Physical Review E* 68.4 (2003), p. 041604.

[26] Anshu Dubey et al. “Structured Adaptive Mesh Refinement Adaptations to Retain Performance Portability With Increasing Heterogeneity”. In: *Computing in Science Engineering* 23.5 (2021), pp. 62–66. DOI: 10.1109/MCSE.2021.3099603.

[27] Blas Echebarria et al. “Quantitative phase-field model of alloy solidification”. In: *Physical Review E* 70 (6 Dec. 2004), p. 061604. DOI: 10.1103/PhysRevE.70.061604.

[28] Robert D. Falgout and Jim E. Jones. “Multigrid on Massively Parallel Architectures”. In: *Multigrid Methods VI*. Ed. by Erik Dick, Kris Riemslagh, and Jan Vierendeels. Berlin, Heidelberg: Springer, 2000, pp. 101–107. ISBN: 978-3-642-58312-4.

[29] Robert D. Falgout, Jim E. Jones, and Ulrike M. Yang. “The design and implementation of hypre, a library of parallel high performance preconditioners”. In: (2006). Ed. by A. M. Bruaset and A. Tveito. Also available as LLNL technical report UCRL-JRNL-205459, pp. 267–294. DOI: 10.1007/3-540-31619-1.

[30] W. R. Fehlner and S. H. Vosko. “A product representation for cubic harmonics and special directions for the determination of the Fermi surface and related properties”. In: *Canadian Journal of Physics* 54.21 (1976), pp. 2159–2169. DOI: 10.1139/p76-256.

[31] R. Fletcher. “Conjugate gradient methods for indefinite systems”. In: *Numerical Analysis*. Ed. by G. Alistair Watson. Berlin, Heidelberg: Springer, 1976, pp. 73–89. ISBN: 978-3-540-38129-7.

[32] *Cyclic reduction – history and applications*. Singapore: Springer, 1997, pp. 73–85.

[33] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996. ISBN: 978-0-801-85414-9.

[34] J. R. Green et al. “An Adaptive, Multilevel Scheme for the Implicit Solution of Three-Dimensional Phase-Field Equations”. In: *Numerical Methods for Partial Differential Equations* 27.1 (Jan. 2011), pp. 106–120. URL: http://eprints.whiterose.ac.uk/78689/.

[35] James E. Guilkey, James B. Hoying, and Jeffrey A. Weiss. “Computational modeling of multicellular constructs with the material point method ”. In: *Journal of Biomechanics* 39.11 (2006), pp. 2074–2086. ISSN: 0021-9290. DOI: 10.1016/j.jbiomech.2005.06.017.

[36] Y. Guo and J. A. Nairn. “Calculation of J-Integral and Stress Intensity Factors using the Material Point Method”. In: *Computer Modeling in Engineering & Sciences* 6.3 (2004), pp. 295–308.

[37] Zhenlin Guo and S. M. Xiong. “On solving the 3-D phase field equations by employing a parallel-adaptive mesh refinement (Para-AMR) algorithm”. In: *Computer Physics Communications* 190 (2015), pp. 89–97. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.01.016.

[38] William D. Henshaw and Donald W. Schwendeman. “Parallel Computation of Three-Dimensional Flows Using Overlapping Grids with Adaptive Mesh Refinement”. In: *J. Comput. Phys.* 227.16 (Aug. 2008), pp. 7469–7502. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2008.04.033.

[39] Magnus R. Hestenes and Eduard Stiefel. “Methods of Conjugate Gradients for Solving Linear Systems”. In: *Journal of Research of the National Bureau of Standards* 49.6 (Dec. 1952), p. 409. DOI: 10.6028/jres.049.044.

[40] J. K. Holmen et al. “Improving Uintah’s Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks”. In: *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact.* PEARC17 27. 2017, 27:1–27:8. ISBN: 978-1-4503-5272-7. URL: http://www.sci.utah.edu/publications/Hol2017b/pearc17.pdf.

[41] J. K. Holmen et al. *Uintah+Hedgehog: Combining Parallelism Models for End-to-End Large-Scale Simulation Performance.* Tech. rep. Scientific Computing and Imaging Institute, 2021. URL: http://www.sci.utah.edu/publications/Hol2021d/UUSCI-2021-002.pdf.

[42] John K. Holmen, Damodar Sahasrabudhe, and Martin Berzins. “A Heterogeneous MPI+PPL Task Scheduling Approach for Asynchronous Many-Task Runtime Systems”. In: *PEARC ’21: Practice and Experience in Advanced Research Computing, Boston, MA, USA, July 18-22, 2021.* Ed. by Joseph Paris et al. ACM, 2021. ISBN: 978-1-4503-8292-2. DOI: 10.1145/3437359.3465581.

[43] Richard D. Hornung and Jeffrey A. Keasler. *The RAJA Portability Layer: Overview and Status.* Tech. rep. 2014.

[44] Richard D. Hornung and Scott R. Kohn. “Managing application complexity in the SAMRAI object-oriented framework”. In: *Concurrency and Computation: Practice and Experience* 14.5 (2002), pp. 347–368. DOI: 10.1002/cpe.652.

[45] A. M. Jokisaari et al. “Benchmark problems for numerical implementations of phase field models”. In: *Computational Materials Science* 126 (2017), pp. 139–151. ISSN: 0927-0256. DOI: 10.1016/j.commatsci.2016.09.022.

[46] Laxmikant V. Kale´ et al. “Programming Petascale Applications with Charm++”. In: *Petascale Computing: Algorithms and Applications*. Ed. by D.A. Bader. 1st. Chapman and Hall/CRC, 2007. Chap. 2. DOI: 10.1201/9781584889106.

[47] Alain Karma. “Phase-Field Formulation for Quantitative Modeling of Alloy Solidification”. In: *Physical Review Letters* 87 (11 Aug. 2001), p. 115701. DOI: 10.1103/PhysRevLett.87.115701.

[48] Alain Karma and Wouter-Jan Rappel. “Phase-field method for computationally efficient modeling of solidification with arbitrary interface kinetics”. In: *Physical Review E* 53 (4 Apr. 1996), R3017–R3020. DOI: 10.1103/PhysRevE.53.R3017.

[49] Alain Karma and Wouter-Jan Rappel. “Quantitative phase-field modeling of dendritic growth in two and three dimensions”. In: *Physical Review E* 57 (4 Apr. 1998), pp. 4323–4349. DOI: 10.1103/PhysRevE.57.4323.

[50] B. Kashiwa. *A multifield model and method for fluid-structure interaction dynamics*. Tech. rep. Los Alamos, NM: Los Alamos National Laboratory, 2001.

[51] B. Kashiwa and E. Gaffney. *Design basis for cfdlib*. Tech. rep. Los Alamos, NM: Los Alamos National Laboratory, 2002.

[52] B. Kashiwa, M. Lewis, and T. Wilson. *Fluid-structure interaction modeling*. Tech. rep. Los Alamos, NM: Los Alamos National Laboratory, 1996.

[53] Michel Kern. “Numerical Methods for Least Squares Problems”. In: *Numerical Methods for Inverse Problems*. John Wiley & Sons, 2016. Chap. Appendix 1, pp. 167–182. ISBN: 978-1-119-13694-1. DOI: 10.1002/9781119136941.app1.

[54] Ronan Keryell, Maria Rovatsou, and Lee Howes. “Khronos Group SYCL 1.2.1 Specification”. In: (2019).

[55] David A. Kessler and Herbert Levine. “Velocity selection in dendritic growth”. In: *Phys. Rev. B* 33 (11 June 1986), pp. 7867–7870. DOI: 10.1103/PhysRevB.33.7867.

[56] Ryo Kobayashi. “Modeling and numerical simulations of dendritic crystal growth”. In: *Physica D: Nonlinear Phenomena* 63.3 (1993), pp. 410–423. ISSN: 0167-2789. DOI: 10.1016/0167-2789(93)90120-P.

[57] G. Krishnamoorthy et al. “Numerical Modeling of Radiative Heat Transfer in Pool Fire Simulations”. In: *ASME proceedings*. Vol. 42215. Heat Transfer, Part A. 2005, pp. 327–337. DOI: 10.1115/IMECE2005-81095.

[58] J. S. Langer. “Models of Pattern Formation in First-Order Phase Transitions”. In: *Directions in Condensed Matter Physics*. Vol. 1. Directions in Condensed Matter Physics. Singapore: World Scientific, 1986, pp. 165–187.

[59] Yibao Li, Hyun Geun Lee, and Junseok Kim. “A fast, robust, and accurate operator splitting method for phase-field simulations of crystal growth”. In: *Journal of Crystal Growth* 321.1 (Apr. 2011), pp. 176–182. ISSN: 0022-0248. DOI: 10.1016/j.jcrysgro.2011.02.042.

[60] H. K. Lin, C. C. Chen, and C. W. Lan. “Adaptive three-dimensional phase-field modeling of dendritic crystal growth with high anisotropy”. In: *Journal of Crystal Growth* 318.1 (2011). The 16th International Conference on Crystal Growth (ICCG16)/The 14th International Conference on Vapor Growth and Epitaxy (ICVGE14), pp. 51–54. ISSN: 0022-0248. DOI: 10.1016/j.jcrysgro.2010.11.013.

[61] Justin Luitjens and Martin Berzins. “Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework”. In: *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS10)*. Atlanta, GA, 2010, pp. 1–10. DOI: 10.1109/IPDPS.2010.5470437.

[62] Justin Luitjens et al. “Uintah Framework”. 2010. URL: %7Bhttp://ccrg.rit.edu/~carpet/images/8/8a/Uintah_Framework.pdf%7D.

[63] Peter MacNeice et al. “PARAMESH: A parallel adaptive mesh refinement community toolkit”. In: *Computer Physics Communications* 126.3 (2000), pp. 330–354. ISSN: 0010-4655. DOI: 10.1016/S0010-4655(99)00501-9.

[64] Stephen F. McCormick. “The Fast Adaptive Composite Grid Method”. In: *Multilevel Adaptive Methods for Partial Differential Equations*. Society for Industrial and Applied Mathematics, 1989. Chap. 4, pp. 81–127. ISBN: 978-0-89871-247-6. DOI: 10.1137/1.9781611971026.ch4.

[65] G. B. McFadden et al. “Phase-field models for anisotropic interfaces”. In: *Physical Review E* 48 (3 Sept. 1993), pp. 2016–2024. DOI: 10.1103/PhysRevE.48.2016.

[66] David S Medina, Amik St-Cyr, and T. Warburton. *OCCA: A unified approach to multi-threading languages*. 2014. DOI: 10.48550/ARXIV.1403.0968.

[67] Daniel I. Meiron. “Selection of steady states in the two-dimensional symmetric model of dendritic growth”. In: *Phys. Rev. A* 33 (4 Apr. 1986), pp. 2704–2715. DOI: 10.1103/PhysRevA.33.2704.

[68] Qingyu Meng et al. “Investigating applications portability with the uintah DAG-based runtime system on petascale supercomputers”. In: *SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12.

[69] H Miura. “Anisotropy function of kinetic coefficient for phase-field simulations: Reproduction of kinetic Wulff shape with arbitrary face angles”. In: *Journal of Crystal Growth* 367 (2013), pp. 8–17. ISSN: 0022-0248. DOI: 10.1016/j.jcrysgro.2013.01.014.

[70] J. Narski and M. Picasso. “Adaptive 3D finite elements with high aspect ratio for dendritic growth of a binary alloy including fluid flow induced by shrinkage”. In: *Fluid Dynamics & Materials Processing* 3.1 (2007), pp. 49–64.

[71] Steven G. Parker. “A component-based architecture for parallel multi-physics {PDE} simulation ”. In: *Future Generation Computer Systems* 22.1–2 (2006), pp. 204–216. ISSN: 0167-739X. DOI: 10.1016/j.future.2005.04.001.

[72] Steven G. Parker, James E. Guilkey, and Todd Harman. “A component-based parallel infrastructure for the simulation of fluid–structure interaction”. In: *Engineering with Computers* 22.3 (2006), pp. 277–292. ISSN: 1435-5663. DOI: 10.1007/s00366-006-0047-5.

[73] R. L. Pego. “Front migration in the nonlinear Cahn-Hilliard equation”. In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 422.1863 (1989), pp. 261–278. ISSN: 0080-4630. DOI: 10.1098/rspa.1989.0027.

[74] Oliver Penrose and Paul C. Fife. “Thermodynamically consistent models of phase-field type for the kinetic of phase transitions ”. In: *Physica D: Nonlinear Phenomena* 43.1 (1990), pp. 44–62. ISSN: 0167-2789. DOI: 10.1016/0167-2789(90)90015-H.

[75] H. Puff. “Contribution to the Theory of Cubic Harmonics”. In: *Physica Status Solidi B* 41.1 (1970), pp. 11–22. DOI: 10.1002/pssb.19700410102.

[76] Alfio Quarteroni and Alberto Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Springer, 1994.

[77] J. C. Ramirez and C. Beckermann. “Examination of binary alloy free dendritic growth theories with a phase-field model”. In: *Acta Materialia* 53.6 (2005), pp. 1721–1736. ISSN: 1359-6454. DOI: 10.1016/j.actamat.2004.12.021.

[78] Luis Reyna and Michael J. Ward. “Metastable Internal Layer Dynamics For The Viscous Cahn-Hilliard Equation”. In: *Methods and Applications of Analysis* 2 (1995), pp. 285–306.

[79] Jan Rosam. “A Fully implicit, fully adaptive multigrid method for multiscale phase-field modelling”. PhD thesis. University of Leeds, Sept. 2007.

[80] Jan Rosam, Peter K. Jimack, and A. M. Mullis. “An adaptive, fully implicit multigrid phase-field model for the quantitative simulation of non-isothermal binary alloy solidification”. In: *Acta Materialia* 56.17 (2008), pp. 4559–4569. ISSN: 1359-6454. DOI: 10.1016/j.actamat.2008.05.029.

[81] Christopher J. Roy. “Review of code and solution verification procedures for computational simulation”. In: *Journal of Computational Physics* 205.1 (2005), pp. 131–156. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2004.10.036.

[82] Youcef Saad. “A Flexible Inner-Outer Preconditioned GMRES Algorithm”. In: *SIAM Journal on Scientific Computing* 14.2 (1993), pp. 461–469. DOI: 10.1137/0914028.

[83] Youcef Saad and Martin H. Schultz. “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. DOI: 10.1137/0907058.

[84] Damodar Sahasrabudhe and Martin Berzins. “Improving Performance of the Hypre Iterative Solver for Uintah Combustion Codes on Manycore Architectures Using MPI Endpoints and Kernel Consolidation”. In: *Computational Science – ICCS 2020*. Ed. by Valeria V. Krzhizhanovskaya et al. Cham: Springer International Publishing, 2020, pp. 175–190. ISBN: 978-3-030-50371-0.

[85] Damodar Sahasrabudhe et al. “A Portable SIMD Primitive Using Kokkos for Heterogeneous Architectures”. In: *Accelerator Programming Using Directives*. Ed. by Sandra Wienke and Sridutt Bhalachandra. Cham: Springer International Publishing, 2020, pp. 140–163. ISBN: 978-3-030-49943-3.

[86] Damodar Sahasrabudhe et al. “Optimizing the hypre solver for manycore and GPU architectures”. In: *Journal of Computational Science* 49 (2021), p. 101279. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2020.101279.

[87] C. Sarkis et al. “Three-dimensional modeling of a thermal dendrite using the phase field method with automatic anisotropic and unstructured adaptive finite element meshing”. In: *IOP Conference Series: Materials Science and Engineering* 117 (Mar. 2016), p. 012008. DOI: 10.1088/1757-899x/117/1/012008.

[88] Steve Schaffer. “A Semicoarsening Multigrid Method for Elliptic Partial Differential Equations with Highly Discontinuous and Anisotropic Coefficients”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 228–242. DOI: 10.1137/S1064827595281587.

[89] Lijian Tan and Nicholas Zabaras. “A level set simulation of dendritic solidification of multi-component alloys”. In: *Journal of Computational Physics* 221.1 (2007), pp. 9–40. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2006.06.003.

[90] X. Tong et al. “Phase-field simulations of dendritic crystal growth in a forced flow”. In: *Physical Review E* 63 (6 May 2001), p. 061601. DOI: 10.1103/PhysRevE.63.061601.

[91] R. Tönhardt and G. Amberg. “Phase-field simulation of dendritic growth in a shear flow”. In: *Journal of Crystal Growth* 194.3 (1998), pp. 406–425. ISSN: 0022-0248. DOI: 10.1016/S0022-0248(98)00687-3.

[92] H. A. van der Vorst. “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 13.2 (Mar. 1992), pp. 631–644. ISSN: 0196-5204. DOI: 10.1137/0913035.

[93] Shun-Lien Wang and Robert F. Sekerka. “Computation of the dendritic operating state at large supercoolings by the phase field model”. In: *Physical Review E* 53 (4 Apr. 1996), pp. 3760–3776. DOI: 10.1103/PhysRevE.53.3760.

[94] Shun-Lien Wang et al. “Thermodynamically-consistent phase-field models for solidification ”. In: *Physica D: Nonlinear Phenomena* 69.1–2 (1993), pp. 189–200. ISSN: 0167-2789. DOI: 10.1016/0167-2789(93)90189-8.

[95] Brian Wetton. *1D Cahn Hilliard Computation (YouTube Video)*. Sept. 2015. URL: https://youtu.be/cq2o2AUUXGM.

[96] Brian Wetton. *2D Allen Cahn Simulation (YouTube Video)*. Mar. 2018. URL: https://youtu.be/6ojleQaCuyE.

[97] Brian Wetton. *2D Cahn Hilliard Energy Simulation (YouTube Video)*. Feb. 2018. URL: https://youtu.be/MovUu2DwWvI.

[98] Brian Wetton. *2D Periodic Cahn Hilliard Simulation (YouTube Video)*. Mar. 2018. URL: https://youtu.be/nKstgHLuQFs.

[99] A. A. Wheeler, B. T. Murray, and R. J. Schaefer. “Computation of dendrites using a phase field model ”. In: *Physica D: Nonlinear Phenomena* 66.1–2 (1993), pp. 243–262. ISSN: 0167-2789. DOI: 10.1016/0167-2789(93)90242-S.

[100] Steven M. Wise, John S. Lowengrub, and Veronica Cristini. “An Adaptive Multigrid Algorithm for Simulating Solid Tumor Growth Using Mixture Models”. In: *Mathematical and Computer Modelling* 53.1–2 (2011), pp. 1–20.

[101] Fengwei W. Yang. *Phase Field Benchmarking (Github Repository)*. 2019. URL: https://github.com/timondy/PhaseFieldBenchmarking.

[102] Fengwei W. Yang et al. “An optimally efficient technique for the solution of systems of nonlinear parabolic partical differential equations”. In: *Advances in Engineering Software* 103 (2017). Special Issue: Civil-Comp: Parallel, Distributed and Cloud Computing, pp. 65–84. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2016.06.003.

[103] A. Younsi and A. Cartalade. “On anisotropy function in crystal growth simulations using Lattice Boltzmann equation”. In: *Journal of Computational Physics* 325 (Nov. 2016), pp. 1–21. DOI: 10.1016/j.jcp.2016.08.014.

[104] Pengtao Yue et al. “A diffuse-interface method for simulating two-phase flows of complex fluids”. In: *Journal of Fluid Mechanics* 515 (2004), pp. 293–317. DOI: 10.1017/S0022112004000370.

[105] Zhang et al. “AMReX: A Framework for Block-Structured Adaptive Mesh Refinement”. In: *Journal of Open Source Software* 4.37 (2019), p. 1370. DOI: 10.21105/joss.01370.

[106] Weiqun Zhang et al. “BoxLib with Tiling: An Adaptive Mesh Refinement Software Framework”. In: *SIAM Journal on Scientific Computing* 38.5 (2016), S156–S172. DOI: 10.1137/15M102616X.

[107] P. Zhao, J. C. Heinrich, and D. R. Poirier. “Dendritic solidification of binary alloys with free and forced convection”. In: *International Journal for Numerical Methods in Fluids* 49.3 (2005), pp. 233–266. ISSN: 1097-0363. DOI: 10.1002/fld.988.

Index

Appendices

Appendix A

Computing Tip Characteristics form PureMetal Solutions

Tip position, velocity and curvatures are all quantities of physical interest that usually are not explicitly computed during a simulation, but later by post-processing the simulation output.

When required, they must be evaluated from the grid values of the phase field, $\varphi_{\boldsymbol{i}}$, either during the simulation or post processing the output.

Since the output produced by Uintah can be large, it has been chosen to allow the computation of these quantities during the simulations itself. When the user requests it in the problem specification file, the tasks for the computation of these quantities are scheduled at the end of each timestep and files are created. These files store the values of the computed position, velocity and curvature at every time step, independently from the output frequency of grid variables.

Different algorithms have to be implemented according to the direction along which the dendrite growth is favored. This direction is determined by the sign of the anisotropy strength. When $\varepsilon > 0$, the arm growth is favored along the cartesian axes (this case is referred as the parallel case) and the algorithm for computing the tip quantities focuses on the arm along the positive x semi-axis; otherwise the dendrite arms grow along the bisectors (or trisectors) (also diagonal case) and the algorithm focuses on the arm growing along $x = y(= z)$.

The diagonal growth case present the advantage of requiring a smaller computational domain but introduces some complications in the computation of the tip position, velocity, and curvature.

For each of these cases, two different approaches have been considered: the first generalizes the approach in [49] and alternates unidimensional polynomial approximations in the directions parallel and orthogonal to the arm growth; the second approach uses bidimensional non linear regression to fit the computed phase field grid values.

In two dimensions, the tip position at a given time, $t \geq 0$, is so defined as the positive time-dependent scalar quantity $r_{\text{tip}}(t) > 0$ such that

$$\psi(r_{\text{tip}}\mathbf{e}_{\text{arm}}, t) = 0, \qquad t > 0\,,$$

where $\mathbf{e}_{\text{tip}}$ denotes the arm direction: $\mathbf{e}_{\text{tip}} = (1,0)^t$ in the parallel case and $\mathbf{e}_{\text{tip}} = \frac{1}{\sqrt{2}}(1,1)^t$ in the diagonal one. The tip velocity is then the derivative of the tip position

$$v_{\text{tip}} := r'_{\text{tip}}\,.$$

The following expression of the curvature for implicit surfaces can be used to compute the curvature at the solid/liquid interface ($\psi = 0$):

$$\kappa = \frac{-\psi_x^2\psi_{yy} + \psi_x\psi_y\psi_{xy} - \psi_y^2\psi_{xx}}{(\psi_x^2 + \psi_y^2)^{3/2}}\,, \tag{A.1}$$

where the pedices are used to denote first and second order spatial derivatives in the Cartesian coordinate directions.

To estimate the tip curvature, it is then necessary either to approximate the derivatives in (A.1) using the discrete values of the phase field computed during the simulation to build multiple univariated polynomial regressors to interpolate the tip and arm characteristics, or to build a single bivariate non-linear regressor and then compute analytically its derivatives; in both approaches it is fundamental to have the most accurate estimate of the tip position r_{tip}.

A.1 The Polynomial Approach

The following *polynomial approach*, which generalizes the algorithm proposed by Karma and Rappel [49], is of the first kind.

A.1.1 Parallel Arm Growth

If we focus to the parallel case and omit the time dependency, we can assume ψ to be even with respect to y

$$\psi(x, y; t) = \psi(x, -y; t)\,. \tag{A.2}$$

Far from the tip the liquid/solid interface of each arm is observed to be approximable by a parabola with the axis parallel to the favored direction of growth, that is there exists a region $\mathcal{I}_{\text{arm}}$ over which it is possible to define a function $g : \mathbb{R} \to \mathbb{R}$ such that

$$\psi(g(y), y) \equiv 0, \qquad (g(y), y)^t \in \mathcal{I}_{\text{arm}}\,,$$

and $g(y) \approx g_{\text{arm}}(y) = ay^2 + by + c$. Because of (A.2), the coefficient b has to be zero and the parameters a and c can be estimated using linear regression over an opportune set of zeros of the function ψ that belong to $\mathcal{I}_{\text{arm}}$. The set $\mathcal{I}_{\text{arm}}$ has to be be a subset of the region between the x axis and the bisector of the first quadrant in order to use only the portion of the liquid/solid interface that belongs to the arm growing in the positive x direction that has positive ordinate. Moreover we have to ensure that g is well defined, therefore we have to choose $\mathcal{I}_{\text{arm}}$ to not contain any point where $\psi_x = 0$.

At the tip, $x = r_{\text{tip}}$, $y = 0$ and $\psi_y = 0$, therefore the *tip curvature* expression (A.1) simplifies to

$$\kappa_{\text{tip}} = -\frac{\psi_{yy}(r_{\text{tip}}, 0)}{\psi_x(r_{\text{tip}}, 0)} \tag{A.3}$$

Given the discrete solution over the first quadrant at the n-th timestep $\boldsymbol{\psi}^n = (\psi^n_{ij} : i, j = 0, 1, \dots M)$ (for AMR simulations we use the only values from the most refined grid and suppose this is wide enough to contain all required grid entries), the first step is to find the index i_{tip} at which the arm tip is located. This could be found by searching for the index, $i_{\text{tip}} \in \{0, 1, \dots, M-1\}$, that satisfies the following condition[1]:

$$\psi^n_{i_{\text{tip}},0}\, \psi^n_{i_{\text{tip}}+1,0} \leq 0\,.$$

In practice, we first look for all the locations next to the solid/liquid interface:

$$\mathscr{L}^n_{\text{arm}} := \{\boldsymbol{l}^n_i = (i, j_i) \in \mathbb{N}^2 \;:\; \psi^n_{i,j_i}\, \psi^n_{i,j_i+1} \leq 0 \,\wedge\, i < m \,\wedge\, j_i \leq i\}\,.$$

It is safe to assume that only one such location $\boldsymbol{l}^n_i$ for each abscissa index can be found; this justifies using i to index them. The first condition ensures that $\psi_{|x=x_i}$ has a zero in the interval $(y_{j_i}, y_{j_i+1}]$, while the second and third conditions that only locations belonging the arm growing in the positive x direction and with positive ordinates are considered. In order to ensure that also $\psi_x = 0$, a location $\boldsymbol{l}^n_i$ in $\mathscr{L}^n_{\text{arm}}$ has to be discarded if exists $\boldsymbol{l}^n_{i+1} \in \mathscr{L}^n_{\text{arm}}$ and $\boldsymbol{l}^n_{i+1} \geq \boldsymbol{l}^n_i$

$$\mathscr{L}^{n\star}_{\text{arm}} := \{\boldsymbol{l}^n_i \in \mathscr{L}^n_{\text{arm}} : \boldsymbol{l}^n_{i+1} \in \mathscr{L}^n_{\text{arm}} \Rightarrow j_{i+1} < j_i\}\,.$$

The tip location is then computed as the maximum index for which such a location exists.

The second step is to estimate both the tip position, r_{tip}, and the derivative $\psi_x(r_{\text{tip}}, 0)$. Chosen $n_2 \geq 2$, let F_j be an approximation of $\psi_{|y=y_j}$ for $j = 0, \dots, n_2 - 1$. In particular, let F_j be the polynomial of degree $d_0 > 0$ that fits the $n_0 > d_0$ points $(x_i, \psi^n_{i,j})$, $i_{\text{tip}} - \lfloor \frac{n_0}{2} \rfloor \leq i < i_{\text{tip}} + \lceil \frac{n_0}{2} \rceil$. In Figure A.1 these points are colored in green. The details of the polynomial fitting algorithm used hereafter and of its implementation are available in [53].

We then define z_j $(j = 0, \dots, n_2{-}1)$ to be the zero of F_j in the interval $[x_{i_{\text{tip}} - \lfloor \frac{n_0}{2} \rfloor}, x_{i_{\text{tip}} + \lceil \frac{n_0}{2} \rceil - 1}]$

[1]the value $\psi^n_{i_{\text{tip}}+1,0}$ if well defined also when $i = M-1$ even if the index $i+1$ refers to a location outside the computational domain thanks to the use of ghost nodes.

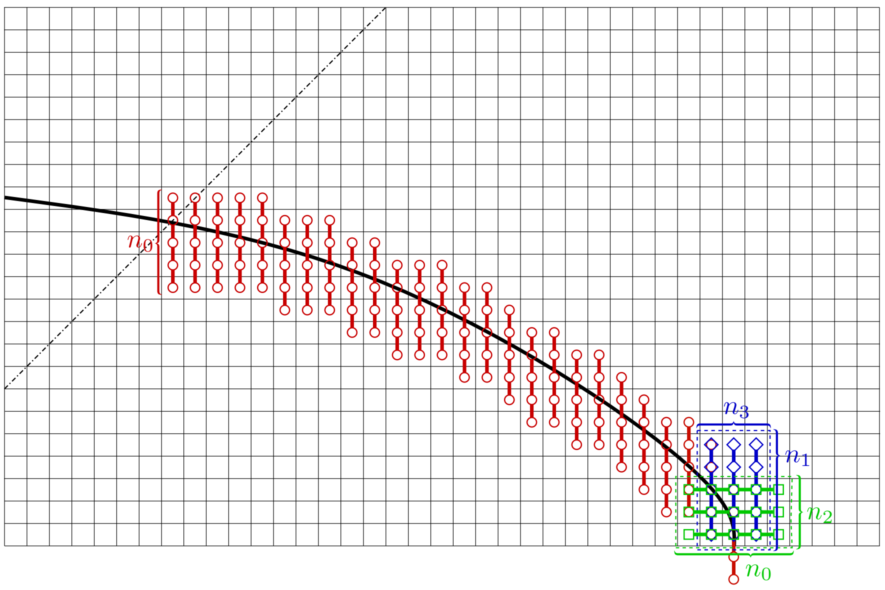


Figure A.1: Cell centred polynomial interpolation of the tip characteristic in the parallel case.

and dF_j to be its derivative at z_j

$$F_j(z_j) = 0\,, \qquad\qquad dF_j = F_j'(z_j)\,.$$

These values are then fitted to approximate $g \circ h$ and $\psi_{x|x=g(y)} \circ h$ close to the tip, where the transformation $h(y) := y^2$ is introduced, since (A.2), to enforce to approximation to be even with respect to t. Therefore, let g_{tip} be the polynomial of degree $0 < d_2 < n_2$ that fits the n_2 points (y_j^2, z_j), and F_x be the d_2-degree polynomial regressor of the n_2 points (y_j^2, dF_j). Evaluating these regressors at $y^2 = 0$ it is then possible to approximate the tip position and ψ_x at the tip:

$$r_{\text{tip}}(t^n) \approx r^n_{\text{tip}} := g_{\text{tip}}(0)\,, \qquad \psi_x(r_{\text{tip}}, 0) \approx F_x(0), \qquad n = 0, 1, \ldots\,.$$

The tip velocity is estimated immediately using a backward finite-difference

$$v_{\text{tip}}(t^n) \approx v^n_{\text{tip}} := \frac{r^n_{\text{tip}} - r^{n-1}_{\text{tip}}}{k}, \qquad n = 1, 2, \ldots\,. \tag{A.4}$$

An approximation $\psi_{yy}(r_{\text{tip}}, 0)$ is still needed to estimate (A.3). Similar to the approximation of $\psi_x(r_{\text{tip}}, 0)$, let G_i be an approximation of $\psi_{|x=x_i} \circ h$ for $i_{\text{tip}} - \lfloor \frac{n_3}{2} \rfloor \le i < i_{\text{tip}} + \lceil \frac{n_3}{2} \rceil$, where $n_3 \ge 2$ is an arbitrary integer. In particular let G_i be the polynomial of degree $d_1 > 0$ that fits the $n_1 > d_1$ points (y_j^2, ψ^n_{ij}), $(j = 0, \ldots n_1 - 1)$. These points are shown in green in Figure A.1. An approximation G_{yy} of $\psi_{yy|y=0}$ is then computed fitting the ordinates $ddG_i := G_i''(0)[h'(0)]^2 +$

$G_i'(0)h''(0) = 2\bar{G}_i'(0)$ at the abscissae x_i ($i = 0, \dots, n_3 - 1$) with a polynomial of degree $0 < d_3 < n_3$. The expression for ddG_i is obtained by applying the Faà di Bruno's formula in the computation of the second order derivative of $\psi_{|x=x_i} \circ h$.

The *tip curvature* is then evaluated thanks to these approximations using the following approximation

$$\kappa^n_{\text{tip}} := -\frac{G_{yy}(r_{\text{tip}})}{F_x(0)} \,. \tag{A.5}$$

For computing the *arm curvature* (or *parabolic curvature*) we could use location sets introduced earlier $\mathscr{L}^{n\star}_{\text{arm}}$, but we may want as well to exclude locations close to the tip, where the arm shape departs from being parabolic.

Then for $\boldsymbol{l}_i \in \mathscr{L}^{n\star}_{\text{arm}}$ we denote with g_i the polynomial of degree d_0 that approximates $\psi_{|x=x_i}$ by fitting the n_0 points with the given ascissa index, i, closest to the location $\boldsymbol{l}_i = (i, j_i)$: (y_j, ψ_{ij}) with $j_i - \lfloor \frac{n_0}{2} \rfloor \le j < j_i + \lceil \frac{n_0}{2} \rceil$. These points are drawn in red in Figure A.1. The root z_i of g_i that lies in the interval $[y_{j_i - \lfloor \frac{n_0}{2} \rfloor}, y_{j_i + \lceil \frac{n_0}{2} \rceil - 1}]$ is then computed for each location $\boldsymbol{l}_i \in \mathscr{L}^{n\star}_{\text{arm}}$.

At last, the linear regressor, $f_\star : y \mapsto a_\star y + c_\star$, fitting the values x_i at the points z_i^2. The parabola approximating the arm is then

$$g^{n\star}_{\text{arm}} := a_\star y^2 + c_\star \,.$$

Its curvature at its vertex, $(y = 0, x = c_\star)$, is easily computed

$$\kappa^{n\star}_{\text{arm}} = \frac{g^{n\star\prime\prime}_{\text{arm}}(0)}{(1 + g^{n\star\prime}_{\text{arm}}(0))^{3/2}} = 2a_\star \tag{A.6}$$

To identify which locations are close enough to the tip that the arm profile there is not parabolic we can check if the following finite difference is less than a given threshold $\alpha > 0$

$$\frac{|z^2_{i+1} + z^2_{i-1} - 2z^2_i|}{h^2} < \alpha$$

that is that the second order derivative of the arm profile $g \circ h$ with respect to h is bounded.

Therefore, the linear regression, $f_{\star\star} : y \mapsto a_{\star\star} y + c_{\star\star}$, fitting only values relative to the following location set

$$\mathscr{L}^{n\star\star}_{\text{arm}} := \{\boldsymbol{l}^n_i \in \mathscr{L}^{n\star}_{\text{arm}} \ : \ \exists \boldsymbol{l}^n_{i\pm1} \in \mathscr{L}^n_{\text{arm}} \wedge |z^2_{i+1} + z^2_{i-1} - 2z^2_i| < \alpha h^2\} \,.$$

will produce a better approximation of the parabolic profile of the dendritic arm and therefore a better estimate of the parabolic curvature

$$\kappa_{\text{arm}}^{n\star\star} = 2a_{\star\star}\,. \tag{A.7}$$

A.1.2 Diagonal Arm Growth

The polynomial approach described above for the parallel growth case has to be adapted for the diagonal case. Let us introduce a new coordinate system, (r, s), where $r = x + y$ and $s = x - y$, so that we can focus on the dendritic arm growing in along the positive r directions. The assumption (A.2) has to be replaced with the following equation that express the symmetry of the phase field with respect to the axis $s = 0$:

$$F(r, s) = F(r, -s)\,. \tag{A.8}$$

The change in the variables of differentiation in (A.1) gives the following expression for the curvature of the implicit curve $\psi = 0$

$$\kappa = \frac{-\psi_r^2(\psi_{rr} + 3\psi_{ss}) + 4\psi_r\psi_s\psi_{rs} - \psi_s^2(3\psi_{rr} - \psi_{ss})}{[2(\psi_r^2 + \psi_s^2)]^{3/2}} \tag{A.9}$$

At the tip, $r = r_{\text{tip}}$, $s = 0$ and $\psi_s = 0$ so

$$\kappa_{\text{tip}} = -\frac{\psi_{rr}(r_{\text{tip}}, 0) + 3\psi_{ss}(r_{\text{tip}}, 0)}{2^{3/2}\psi_r(r_{\text{tip}}, 0)} \tag{A.10}$$

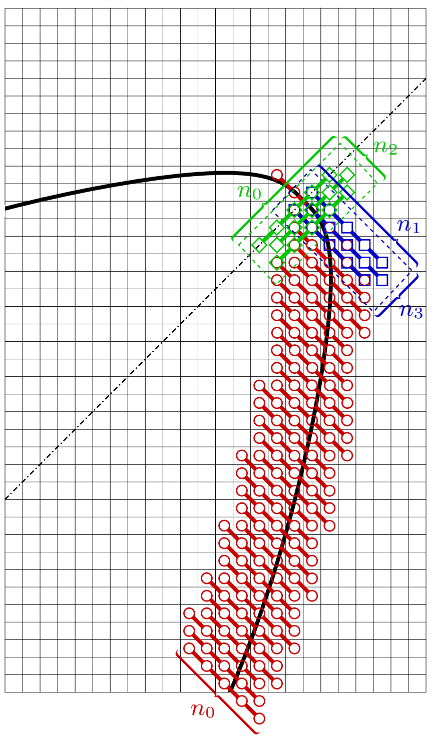
The first step is again to find the location of the arm tip. For this reason we introduce an indexing that is coherent with the new coordinate system $(i, j) \to (\rho = i + j, \varsigma = i - j)$. The tip location is then the maximum index ρ for which an entry exists in the locations next to the liquid/solid interface

$$\mathscr{L}_{\text{arm}}^{n\star} := \{\boldsymbol{l}_\rho^n \in \mathscr{L}_{\text{arm}}^n : \boldsymbol{l}_{\rho+2}^n \in \mathscr{L}_{\text{arm}}^n \Rightarrow \varsigma_{\rho+2} < \varsigma_\rho\}\,,$$

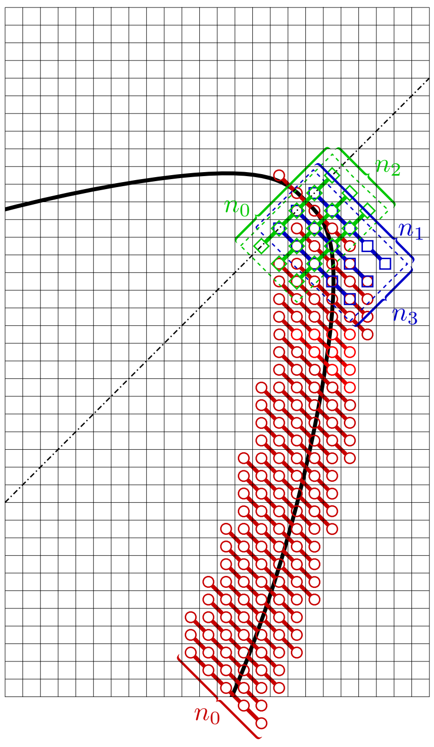
where

$$\mathscr{L}_{\text{arm}}^n := \{\boldsymbol{l}_\rho^n = (\rho, \varsigma_\rho) \in \mathbb{N}^2 \;:\; \psi_{\rho,\varsigma_\rho}^n \psi_{\rho,\varsigma_\rho+1} \le 0 \wedge \rho < 2m \wedge \varsigma_\rho \le \min\{\rho, 2m - \rho\}\}\,.$$

The definition of the last set of location is almost identical to the one used for the parallel case where the indices have been replaced with the newly defined ones and their ranges have been updated accordingly. In the definition of $\mathscr{L}_{\text{arm}}^{n\star}$, on the contrary, the condition for ensuring the monotonicity of the arm profile had to be corrected since $\varsigma_\rho > \varsigma_{\rho+1}$ does not imply that $s_\rho > s_{\rho+1}$, where s_ρ is the zero of $\psi_{|r=r_\rho}$ closest to z_ρ and (r_ρ, z_ρ) are the coordinates of the location $\boldsymbol{l}_\rho$. This is due to the fact that only grid elements with indices of the same parity are aligned.



(a) even and odd



(b) only even

Figure A.2: Cell centred polynomial interpolation of the tip characteristic in the diagonal case.

The second step is to estimate both the tip position, r_{tip}, and the derivative $\psi_r(r_{\text{tip}}, 0)$. As for the parallel case, we want to use n_2 approximation of $\psi_{|s=s_\varsigma}$ for $\varsigma = 0, 1 \ldots, n_2 - 1$ or for $\varsigma = 0, 2 \ldots, n_2 - 1$. The two different choices for the ς index are shown in Figure A.2, both *even and odd* values on the left and only even values on the right.

These approximations, let F_ς are polynomial of degree d_0 computed by fit the n_0 values discrete values of $\boldsymbol{\psi}^n$. In Figure A.2 these points are colored in green.

The values z_ς and dF_ς are defined, as before, as the closest zero to the tip of F_ς and as its derivative at z_ς. In addition to these values, since in (A.10) appears also the term $\psi_{rr}(r\_tip, 0)$, we define also $ddF_\varsigma := F_\varsigma(z_\varsigma)$.

These values are then fitted to approximate $g \circ h$, $\psi_{r|r=g(s)} \circ h$ and $\psi_{rr|r=g(s)} \circ h$ with the d_2-degree polynomials g_{tip}, F_r and F_{rr}. Here g is the graph of ψ with respect to s close to tip: $\psi(g(s), s) \equiv 0$ in an opportune neighbor of the tip.

Evaluating these regressors at $s^2 = 0$ it is then possible to approximate the tip position r_{tip} and the derivatives of the phase field in the r direction at the tip:

$$r_{\text{tip}}(t^n) \approx r^n_{\text{tip}} := g_{\text{tip}}(0)\,, \qquad \psi_r(r_{\text{tip}}, 0) \approx F_r(0)\,, \qquad \psi_{rr}(r_{\text{tip}}, 0) \approx F_{rr}(0), \qquad n = 0, 1, \ldots\,.$$

The tip velocity is estimated using the same backward finite-difference (A.4).

To approximate $\psi_{ss}(r_{\text{tip}}, 0)$ we introduce, as before, n_3 approximations G_ρ of $\psi_{|r=r_\rho} \circ h$. These are polynomial of degree d_1 fitting n_1 values of $\boldsymbol{\psi}^n$ each. We have here the same choice between using values with both even and odd indices for ρ or with only even ρs. These entries are shown in green in Figure A.2. A polynomial, G_{ss} of degree d_3 is then fitted to the points (r_ρ, ddG_ρ) to approximate the function $\psi_{ss|s=0}$. The values ddG_i are defined, as above, as $ddG_\rho := 2\bar{G}'_\rho(0)$.

The *tip curvature* is then evaluated thanks to these approximations using the following approximation

$$\kappa^n_{\text{tip}} := -\frac{F_{rr}(0) + 3G_{ss}(r_{\text{tip}})}{2^{3/2} F_r(0)} \,. \tag{A.11}$$

For computing the parabolic curvatures, as for the parallel case, we denote with g_ρ the polynomial of degree d_0 that approximates $\psi_{|r=r_\rho}$ by fitting the n_0 values of the phase field each location $\boldsymbol{l}_i \in \mathscr{L}^{n\star}_{\text{arm}}$. These points are drawn in red in Figure A.2.

The roots z_ρ of g_ρ closest to the location $\boldsymbol{l}_\rho$ are then computed and linear regression is used to compute the line $f_\star : y \mapsto a_\star y + c_\star$ that fits the values r_ρ at the points z^2_ρ. One parabolic curvature is then compute as $\kappa^{n\star}_{\text{arm}} = 2a_\star$.

Excluding for the set of regression points the locations where the following condition is not satisfied

$$\frac{|z^2_{\rho+1} + z^2_{\rho-1} - 2z^2_\rho|}{2h^2} < \alpha$$

the linear regression, $f_{\star\star} : y \mapsto a_{\star\star} y + c_{\star\star}$, is computed and the second estimate of the parabolic curvature is evaluated as $\kappa^{n\star\star}_{\text{arm}} = 2a_{\star\star}$.

A.2 The Hyperbolic Tangent Approach

As stated previously, an alternative approach to use multiple univariated polynomial regressors for computing the arm profile and the tip and arm characteristics. This tanh approach uses fits the following bivariate function to the same $n_1 \times n_3$ points at the tip that would have been used for the approximation of $\psi_{yy}(r_{\text{tip}}, 0)$, in the parallel case, $\psi_{ss}(r_{\text{tip}}, 0)$, in the diagonal one

$$\hat{\psi}^n(\xi, \zeta) = -\tanh(a + b\xi + c\xi^2 + d\zeta^2)$$

where, depending on the case, either $(\xi, \zeta) \mapsto (x, y)$ or $(\xi, \zeta) \mapsto (r, s)$. The expression of $\hat{\psi}^n$ is similar to the initial conditions (2.10) with an additional term $b\xi$ which takes into account that the focus of the conic approximation of the arm profile at the tip is moving along the axis $\zeta = 0$. The unbounded trust region algorithm used to compute the non linear regressor $\hat{\psi}^n$ is described in [23].

The implicit equation for the arm profile is then a conic with the $\zeta = 0$ as axis of symmetry

$$\frac{a}{d} + \frac{b}{d}\xi + \frac{c}{d}\xi^2 + \zeta^2 = 0\,,$$

where d is assumed to be not equal to zero, otherwise the arm profile would be parallel to one of the ζ axis.

When $c = 0$ this is a parabola and the graph of the arm profile is

$$g(\zeta) = -\frac{a + d\zeta^2}{b}\,.$$

If $c \cdot d < 0$, the conic section is elliptic and its right half graph can be used to describe the arm profile

$$g(\zeta) = -\frac{b}{2c} + \frac{\sqrt{b^2 - 4c(a + d\zeta^2)}}{2|c|}\,;$$

if instead $c \cdot d > 0$, the conic is an hyperbola and its left arm has to be used

$$g(\zeta) = -\frac{b}{2c} - \frac{\sqrt{b^2 - 4c(a + d\zeta^2)}}{2|c|}.$$

The tip position r_{tip} is then

$$r_{\text{tip}} := g(0) = \begin{cases} -\frac{a}{b}\,, & \text{if } c = 0 \\ -\frac{b + \operatorname{sign} d\sqrt{b^2-4ac}}{2c}\,, & \text{otherwise} \end{cases}$$

and, from it, also the tip velocity is computed using (A.4).

In the parallel case, we can substitute $\hat{\psi}^n$ for ψ in (A.3) to obtain the following expression for the tip curvature

$$\kappa^n_{\text{tip}} = -\frac{\hat{\psi}^n_{yy}(r_{\text{tip}}, 0)}{\hat{\psi}^n_x(r_{\text{tip}}, 0)} = -\frac{-2d}{-(b + 2cr_{\text{tip}})} = -\frac{2d}{b + 2cr_{\text{tip}}}\,,$$

while in the diagonal case the same substitution is done in (A.10)

$$\kappa^n_{\text{tip}} = -\frac{\hat{\psi}^n_{rr}(r_{\text{tip}}, 0) + 3\hat{\psi}^n_{ss}(r_{\text{tip}}, 0)}{2^{3/2}\hat{\psi}^n_r(r_{\text{tip}}, 0)} = -\frac{-2(c + 3d)}{-2^{3/2}(b + 2cr_{\text{tip}})} = -\frac{c + 3d}{2^{1/2}(b + 2cr_{\text{tip}})}\,.$$

For computing the parabolic curvatures, the liquid/solid interface is still computed fitting the zeros of functions approximating $\psi_{|x=x_i}$ or $\psi_|r = r_\rho$, according to the growth direction. The same n_0 points used for the polynomial fitting are now used to compute the non linear regressors

$$\hat{g}_{i(\rho)}(\xi) = -\tanh(b\xi + c\xi^2)\,.$$

Their zeros, $z_{i(rho)}$, (or a subset of theirs), are then used to compute the linear regressors $f_{\star(\star)}$ of the points (z_i^2, x) from which the curvatures at the regressor parabola are then evaluated using (A.6) and (A.7).

A.3 Calibration of the Parameters

To find the best combination of post-processing parameters several simulations of the same pure metal problem ($\alpha = 1$, $|\varepsilon| = 0.05$, $r_0 = 16$, $\Delta = 0.65$) on a uniform grid with the same choice of discretization parameters ($h = 0.4$, $k = 0.032$) while changing the post-processing parameters.

Only the first 200 non-dimensional time units of the simulation have been performed for this purpose. This allowed to limit the computational domain to $[0, 60]^2$ using homogeneous Neumann conditions on the $x, y = 0$ edges and Dirichlet conditions on $x, y = 60$.

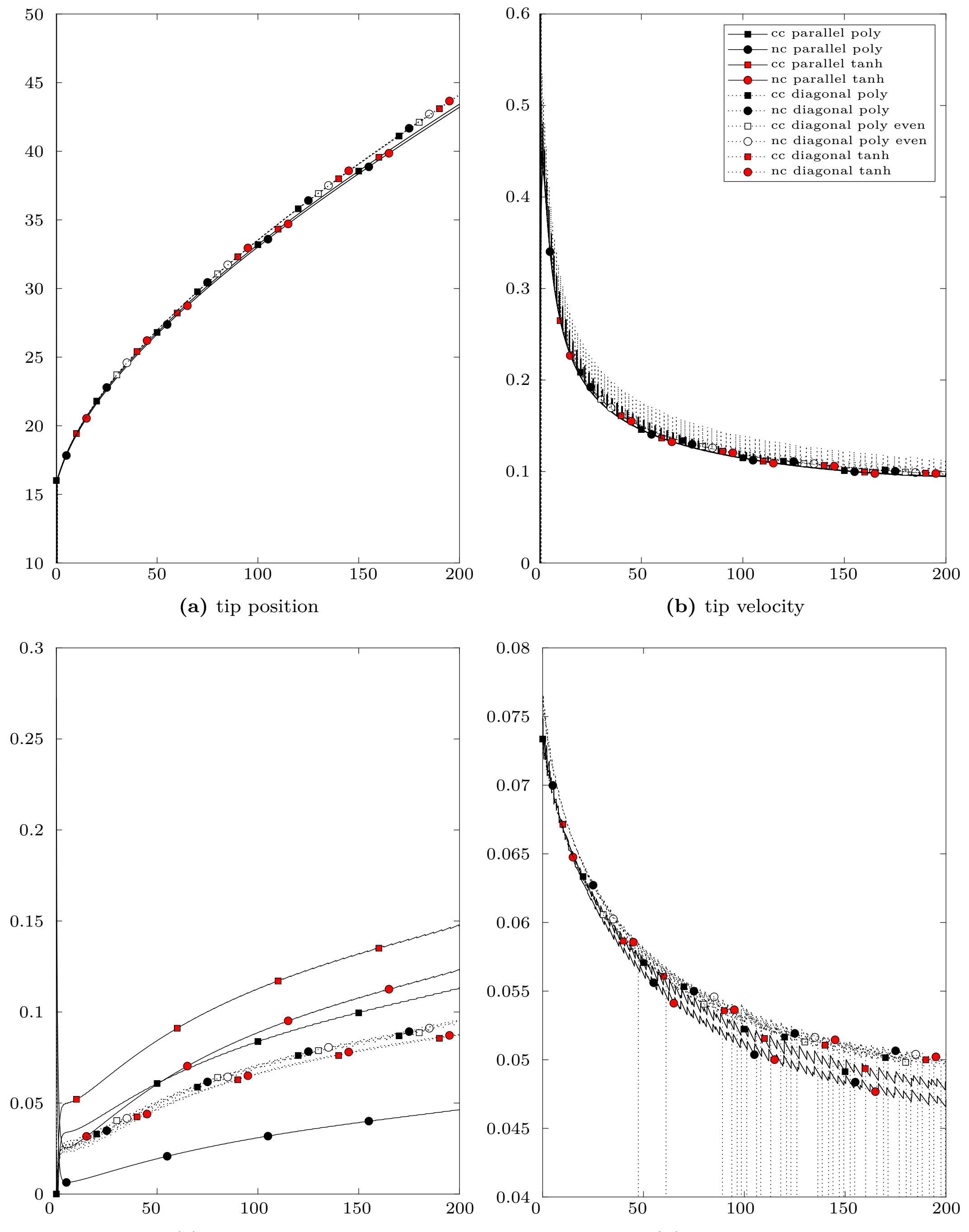
The three algorithms implementing the polynomial approach (parallel case, diagonal case using both even and odd indices, and diagonal case using only even indices) and two implementing the hyperbolic tangent approach (parallel case, and diagonal case using both even and odd indices) are used for simulations using both cell centered and node centered variable bases.

A.3.1 Literature parameters

The polynomial algorithm for the parallel case coincides with the one proposed in [49] when a node centered discretization for ψ is adopted and the following post-processing parameters are chosen: $n_0 = n_3 = 5$, $n_1 = 3$, $n_2 = 1$, $d_0 = d_3 = 4$, $d_1 = 2$, and $d_2 = 0$. When cell centered discretization is adopted, the phase filed has to be interpolated on $y = 0$, therefore n_2 must be greater than 1. When $d_2 = n_2 - 1$ the polynomial regressor is the polynomial interpolator, therefore, for the vertex based case, this choice is equivalent to using only values of the phase field on $y = 0$ ($n_2 = 1$, and $d_2 = 0$). The graphs of the arm characteristics for the choice of post-processing parameters $n_0 = n_2 = n_3 = 5$, $n_1 = 3$, $d_0 = d_2 = d_3 = 4$, and $d_1 = 2$ are shown in Figure A.3.

Looking at the post-processed values of the position (Figure A.3a) grid anisotropy can be observed; the dendritic tip position is growing faster in the diagonal direction (the dashed lines in the plot) as compared with simulations with $\varepsilon > 0$. Moreover, it can be observed that, while all algorithm and choice of variable basis produce similar values for the tip position when $\varepsilon < 0$, the choice between cell centered (square marks) or vertex based (circle marks) discretization affects the position result.

From the analysis of the tip velocity values (Figure A.3b), grid anisotropy is still evident with a grow velocity in the diagonal direction slightly faster than the one in the parallel direction. The choice of variable basis, instead, is not affecting the velocity estimate in neither cases, which means that the differences observed in the computation of the tip position for the parallel case is constant over the simulation. This suggest that this difference is imputable to the post-process



(a) tip position **(b)** tip velocity

(c) tip curvature **(d)** arm curvature

Figure A.3: Tip characteristics computed with literature post-processing parameters ($n_0 = n_2 = n_3 = 5$, $n_1 = 3$, $d_0 = d_2 = d_3 = 4$, and $d_1 = 2$). Only the first 200 non-dimensional units of time of the simulation are performed. The arm curvature is $\kappa_{\text{arm}}^{n\star}$ and is computed fitting all locations in $\mathscr{L}_{\text{arm}}^{n\star}$.

itself. Graphs produced by the hyperbolic tangent approach are discontinuous and shows peaks corresponding to the timesteps where the tip go across a grid position (a cell centroid or a vertex). Graphs produced by the polynomial approach, instead, are smoother and presents very limited steps when the tip is at a grid position only when considering problem with $\varepsilon < 0$.

The tip curvature (Figure A.3c) is the tip characteristic that presents the most variability to the choices of variable basis, favored arm growth direction, and post-processing algorithm. This variability is limited when only values computed from simulations with $\varepsilon < 0$ are considered. In this case, the difference between tip curvature estimations with a different choice of variable basis is negligible and it is also negligible the difference between the values produced by the polynomial approach using both even and odd indices (black markers) and those produced by the polynomial approach using only even indices (white markers). The values estimated by the hyperbolic tangent approach (red markers) are about 6% smaller than the polynomial estimates.

Looking at the arm curvature values (Figure A.3d) it can be observed that this are not smooth and that their variability is higher when the arm growth is favored along the cartesian axes. The arm curvature values, however, are not dependent on the post-processing algorithm and depend only on the choice of variable basis and favored growth direction. Smaller differences are observed between simulations with $\varepsilon < 0$.

A.3.2 Calibration of Parameter n_0

One parameter at a time is modified while keeping the other constant. First, values in the range $2 \div 9$ are considered for the parameter n_0 (for algorithm using the polynomial approach d_0 is modified accordingly so that the polynomial interpolator is chosen as regressor $d_0 = n_0 - 1$).

From the simulations' outputs can be observed that the dependency upon n_0 of the tip position is negligible (less than 0.03%) with the difference between results with consecutive values of n_0 but same ε and post-processing algorithm decreasing as n_0 increases.

This negligible differences, however, result in negligible differences in the tip velocities only when the hyperbolic tangent approach is adopted. In fact for simulation using a polynomial approach it can be observed that using values more than six regression values results in tip velocity profiles that present great irregularities, with a variability that increases as n_0 increases. On the other hand, also using fewer regression points results in less smooth profiles. In particular the profiles present bigger steps corresponding to the timesteps where the tip goes across a grid position as n_0 ranges from 6 to 2. Discontinuities of the velocity profile are observer also for the hyperbolic tangent algorithms which present peaks where the tip goes across a grid position; however the size of this peaks is independent on the choice of n_0 and is about 10% for problems with diagonal arm growth and less than 5% for parallel growth.

Negligible dependency on n_0 for hyperbolic tangent algorithms is observed also in the tip curvature computations. The tip curvature profile for the diagonal cases has big steps ($\sim 65\%$) when $n_0 = 2$. The magnitude of these steps vanishes as the number of regression points increases and almost indistiguishible smooth profiles are obtained with $n_0 \geq 6$. Profiles of parallel problems have a similar dependency on n_0, but the size of the step discontinuities for $n_0 = 2$ is relatively smaller ($\sim 6\%$).

For all choices of post-processing algorithm, variable basis, and favored growth direction no appreciable dependency on n_0 is observed and all the arm curvature profiles present step discontinuities of about 2%.

A.3.3 Calibration of Parameter d_0

From the observations above, $n_0 = 6$ is the best choice for the number of regression points n_0. This allows the parameter d_0 of the polynomial approach to range between 1 and 5. When $d_0 = 1, 2$, the position profile obtained shows step discontinuities of limited magnitude ($\sim 2\%$). Higher values of d_0 results in almost indistinguishable smooth position profiles. Small values of d_0 produce peaks in the velocity profile where the tip goes across a grid position. These peaks are maximum ($\sim 6500\%$) when d_0, decrease as d_0 increases and vanish for $d_0 = 5$. Limited fluctuations ($< 1\%$) in tip curvature profiles are observed for $d_0 = 1, 2$. Fluctuation vanishes as profiles converge to the same profile as d_0 reaches 5.

For diagonal problems also the arm curvature shows a dependency upon the choice of d_0 which however does not affect significantly the magnitude of the step discontinuities.

For the next simulations using a polynomial approach the degree d_0 is therefore chosen to be 5.

A.3.4 Calibration of Parameter n_1

Varying the parameter n_1 in the range $2 \div 9$ result in very limited variations ($< 0.5\%$) in the tip position estimations for all combinations of post-processing algorithm, growth direction and variable basis.

The changes of the parameter n_1 don't affect also the velocity profiles computed with a polynomial approach which don't present discontinuities of any kind. However, when using a hyperbolic tangent fitting, the number of regression points n_1, affects the magnitude of the peaks corresponding to the tip going across a grid position. Their magnitude is slightly higher for cell centered simulations and is about 5% when $n_1 = 2$, is minimal ($\sim 1.5\%$) for $n_1 = 2$, and increases as n_1 increases above 2 reaching a relative value of about 1/4 when $n_1 = 9$.

Greater dependency on n_1 is observed in the tip curvature profiles. All algorithms show the difference between the profiles computed using two adjacent values of n_1 decrease as this number is increased. However the asymptotic profiles to which the different approaches tend are all different in the parallel case. Remembering that n_1 is the number of regression points used to interpolate the phase field over the axis of the favored direction of growth $\zeta = 0$, it is reasonable to believe that this differences should be smaller for diagonal problems and for vertex based variables; in these cases, in fact, grid locations exist on $\zeta = 0$. Moreover, since such locations are more dense in the diagonal case, it could be expected that for this case the differences in the tip curvature profiles are smaller. Smaller differences are in fact observed between simulations of the diagonal problem than between parallel outputs. In addition to this, it can be argued that, while it is improving the smothness of the profiles, increasing n_1 is also making the tip estimation less local. This is corroborated by the fact that in the parallel the profiles with $n_1 = 2$ and the same variable basis overlaps, while they asymptotic profile are different. In the diagonal case as well, tip curvature profiles from different approaches are closer for smaller values of n_1.

The arm curvature is independent of n_1, therefore, from the arguments above, the best choice for n_1 is to set $n_1 = 2$ which implies also the choice $d_1 = 1$ for the polynomial approaches.

A.3.5 Calibration of Parameter n_2

The parameter n_2 appears only in the polynomial approaches and affects appreciably only the tip curvature profile of parallel problems using cell centered variables. In particular, increasing the value of n_2 decreases the difference between subsequent profiles. However the difference between the profile with $n_2 = 2$ and $n_2 = 9$ is less than 0.001%. In the following analysis, therefore, the choices $n_2 = 2$ and $d_2 = 1$ are made.

A.3.6 Calibration of Parameter n_3

Varying n_3 in the range $2 \div 9$ affect only the tip velocity and curvature profiles. Negligible differences between position profile are observed with the parallel approaches. The hyperbolic tangent approach for parallel problems is unstable when $n_3 = 2$. Peaks at grid positions are otherwise observed for all other simulation using the hyperbolic tangent approach. Their magnitude is minimal when $n_3 = 3$.

With regards to the tip curvature profiles, the polynomial interpolation is found unstable for $n_3 \geq 8$. Setting $n_3 = 2$ produces fluctuation in the tip curvature profiles in the order of 2% for parallel simulations using both the polynomial and the hyperbolic tangent approaches. The profile for parallel simulations are otherwise almost undistinguished. The fluctuation found in the tip curvature profile for diagonal polynomial simulations have a magnitude of about 5% that is independent on n_3, when both even and odd regression locations are used and that vanishes when n_3 is increased. In this last case, most of the fluctuations is removed already when $n_3 = 5$ when the fluctuations' amplitude is less then 0.3%. Step discontinuities are present in the tip curvature profiles of diagonal simulations using the hyperbolic tangent approach with an amplitude ($\sim 5\%$) that is independent of n_3.

The arm curvature profile is in any case independent of n_3, therefore the best choice for n_3 appears to be $n_3 = 5$ and the best polynomial approach for diagonal problems is the one using only even locations. This choice for n_3 allows d_3 to range between 1 and 4. Only the tip curvature profile is affected by this choice: profile with $d_3 = 1$ have steps about 10% wide that are reduced to about 1% for $d_3 = 4$.

A.3.7 Calibration of Parameter α

The last parameter to calibrate is the threshold used α for identify which locations are too close to the tip to be used as regression locations in the computation of the arm curvature. For this reason the following values of α have been considered: $\alpha = 1, 0.5, 0.25, 0.125$. The difference between subsequent profiles decreases as α is halved, therefore the value $\alpha = 0.125$ is used hereafter.

A.3.8 Default parameters

From the discussion above, best for the polynomial approach is the following choice of parameters

$$n_0 = 6\,, \quad d_0 = 5\,, \quad n_1 = n_2 = 2\,, \quad d_1 = d_2 = 1\,, n_3 \quad = 5\,, d_3 \quad = 4\,, \alpha \quad = 0.125\,,$$

and using only even locations when $\varepsilon < 0$. Moreover, the best choice of parameter for the hyperbolic tangent approach is

$$n_0 = 6\,, \qquad n_1 = 2\,, \qquad n_3 = 5\,, \qquad \alpha = 0.125\,.$$

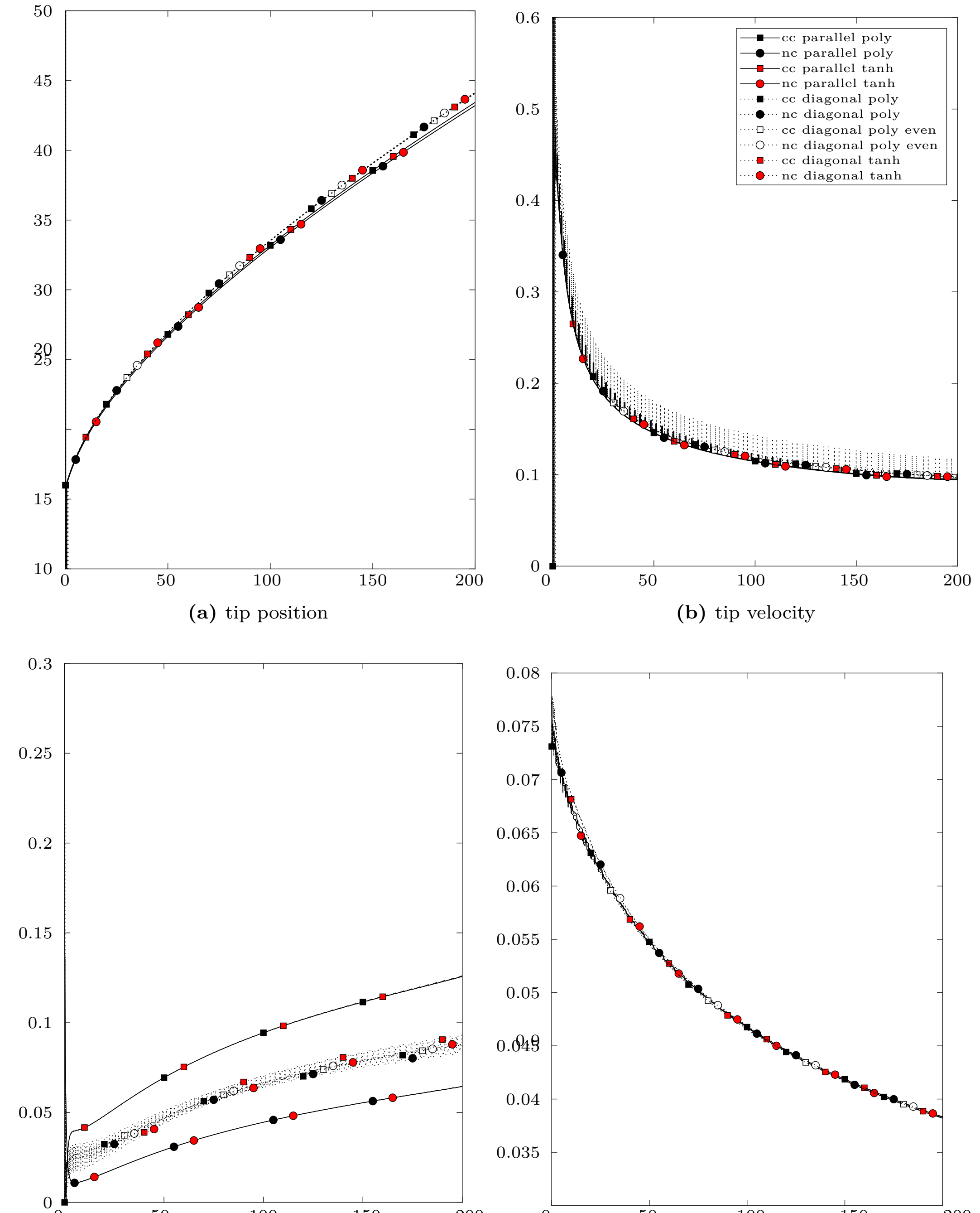
The graphs of the arm characteristics for these choices of post-processing parameters are shown in Figure A.4.

Looking at the position profiles (Figure A.4a) grid anisotropy is still observed, since such behavior is related to the coarseness of the computational grid rather than on the post-process itself. The small difference between the positions computed from cell centered and vertex based simulations with $\varepsilon > 0$ is still appreciable and is not dependent on the post-processing approach and parameters, so it is a behavior that can be considered depending only on the choice of variable basis.

The tip velocity profiles (Figure A.4b) are affected by grif anisotropy as well, with diagonal profiles slightly faster than the parallel ones

Graphs produced by the hyperbolic tangent approach show peak discontinities at timesteps where the tip go across a grid position. Profiles computed with the polynomial approach, instead, are smooth.

Great improvement are found in the tip curvature profiles (Figure A.4c) when compared to ones with the initial choice of parameters (Figure A.3c. Both the polynomial and the hyperbolic tangent approach produce the same profile for the parallel case, which are dependent, as the tip position, only on the variable basis. In the diagonal case, all approaches produce similar profiles independently of the variable basis. These profile have the same average trend but present different variability: using the hyperbolic tangent approach produces profile with step-like discontinities while the polynomial approach produces profiles with fluctuations. The smaller variability ($< 1\%$) is observed when using only even indices with the polynomial approach.



(a) tip position **(b)** tip velocity **(c)** tip curvature **(d)** arm curvature

Figure A.4: Tip characteristics computed with default post-processing parameters ($n_0 = 6$, $n_1 = n_2 = 2$, $n_3 = 5$, $d_0 = 5$, $d_1 = d_2 = 1$, and $d_3 = 4$). Only the first 200 non-dimensional units of time of the simulation are performed. The arm curvature is $\kappa_{\mathrm{arm}}^{n\star\star}$ and is computed excluding locations close to the tip ($\alpha = 0.125$) by fitting locations in $\mathscr{L}_{\mathrm{arm}}^{n\star\star}$.

Also the arm curvature profiles (Figure A.4d) are greatly improved by calibrating α. The profiles obtained are smoother and almost undistinguished between different problems and approaches.

A.4 Computing Tip Characteristics for 3D Problems

When considering three dimensional problems, the arm morphology can be computed as for the two dimensional cases given that we restrict our computations on a planar section of the dendrite that contains one of the lines joining the dendrite center and one of its tips. This can be the plane $x = 0$ or $x = y$ depending on the sign of the anisotropy coefficient ε. By doing so, the same exact algorithms described for 2D problems can be used except for the simulations using cell-centered variables' representation where the growth is favored along the cartesian axes (parallel case). In fact in these simulation the plane $x = 0$ is not identifiable with a layer of cells since their centroid are either on one or the other side of the plane. For such simulations, therefore, computing the characteristics of the dendritic arm and tip would require interpolation or fitting also in the direction orthogonal to the plane.