Asynchronous Stabilisation and Assembly Techniques for Additive Multigrid

CHARLES DAVID MURRAY

Asynchronous Stabilisation and Assembly Techniques for Additive Multigrid

Charles D Murray

A Thesis presented for the degree of
Doctor of Philosophy

Department of Computer Science
Durham University
United Kingdom

June 2021

Asynchronous Stabilisation and Assembly Techniques for Additive Multigrid

Charles D Murray

Submitted for the degree of Doctor of Philosophy
June 2021

Abstract: Multigrid solvers are among the best solvers in the world, but once applied in the real world there are issues they must overcome. Many multigrid phases exhibit low concurrency. Mesh and matrix assembly are challenging to parallelise and introduce algorithmic latency. Dynamically adaptive codes exacerbate these issues. Multigrid codes require the computation of a cascade of matrices and dynamic adaptivity means these matrices are recomputed throughout the solve. Existing methods to compute the matrices are expensive and delay the solve. Non-trivial material parameters further increase the cost of accurate equation integration. We propose to assemble all matrix equations as stencils in a delayed element-wise fashion. Early multigrid iterations use cheap geometric approximations and more accurate updated stencil integrations are computed in parallel with the multigrid cycles. New stencil integrations are evaluated lazily and asynchronously fed to the solver once they become available. They do not delay multigrid iterations. We deploy stencil integrations as parallel tasks that are picked up by cores that would otherwise be idle. Coarse grid solves in multiplicative multigrid also exhibit limited concurrency. Small coarse mesh sizes correspond to small computational workload and require costly synchronisation steps. This acts as a bottleneck and delays solver iterations. Additive multigrid avoids this restriction, but becomes unstable for non-trivial material parameters as additive coarse grid levels tend to overcorrect. This leads to oscillations. We propose a new additive variant, adAFAC-x, with a stabilisation parameter that damps coarse grid corrections to remove oscillations. Per-level we solve an additional equation that produces an auxiliary correction. The auxiliary correction can be computed additively to the rest of the solve and uses ideas similar to smoothed aggregation multigrid to anticipate overcorrections. Pipelining techniques allow adAFAC-x to be written using single-touch semantics on a dynamically adaptive mesh.

Declaration

The work in this thesis is based on research carried out in the Department of Computer Science at Durham University. No part of this thesis has been submitted elsewhere for any degree or qualification.

Note on Publications Included in this Thesis

At the time of submission, this thesis contains chapters that have been based on previously published papers in peer-reviewed journals:

C. D. Murray and T. Weinzierl, 'Stabilized asynchronous fast adaptive composite multigrid using additive damping', *Numerical Linear Algebra with Applications*, 2020, doi:https://doi.org/10.1002/nla.2328

C. D. Murray and T. Weinzierl, 'Lazy stencil integration in multigrid algorithms', in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2019, pp. 25–37, doi:https://doi.org/10.1007/978-3-030-43229-4_3

C. D. Murray and T. Weinzierl, 'Delayed approximate matrix assembly in multigrid with dynamic precisions', *Concurrency and Computation: Practice and Experience*, 2020, doi:https://doi.org/10.1002/cpe.5941

Acknowledgements

This PHD was made possible by a Durham University/EPSRC DTA PhD scholarship. This work has also made use of the Hamilton HPC Service of Durham University. The source code used to support this thesis can be found at:
https://bitbucket.org/CDMurray/adafacx/src/master/

Thanks to my supervisor Tobias Weinzierl for his continued support and help throughout my PhD. His knowledge and hands on approach has been invaluable—without him none of this would have been possible. I am incredibly grateful for his aid.

I would also like to thank my friends and family for all their support through my PhD—especially my sister for being willing to proofread my thesis even though this thesis is most definitely not within her field. Thanks are also due to the metalcore band Converge, for providing the inspiration of appropriate behaviour for multigrid algorithms.

Contents

List of Figures

List of Tables

Summary of Nomenclature

u_ℓ	Solution on level ℓ
b_ℓ	Right-hand side on level ℓ
A_ℓ	Matrix on level ℓ
M_ℓ^{-1}	Smoother (cheap approximation of A_ℓ) on level ℓ
$\tilde{M}_{\ell-1}^{-1}$	Smoother on an auxiliary equation level $\ell - 1$
$R_{\ell+1}^{\ell}$	Restriction from fine grid $\ell + 1$ to level ℓ
$P_{\ell}^{\ell+1}$	Prolongation from level ℓ to fine grid $\ell + 1$
$\ell_{\max}$	Level index for the finest grid
$\mathcal{S}$	Multigrid smoother iteration across all levels
$\mathcal{S}_\ell$	Multigrid smoother iteration on single grid level
$\mathcal{S}^{DoF}$	Individual smoothing update of the multigrid algorithm
$\mathcal{A}_\ell$	Generic assembly process for level ℓ
$\mathcal{A}_\ell^{(\text{geo})}$	Geometric assembly process for level ℓ
$\mathcal{A}^{(\text{geo})}(n)$	Geometric assembly process for level ℓ with n sampling points
n	number of subcells divisions in each dimension
$\mathcal{A}^{(\text{alg})}$	Algebraic coarse grid assembly process
$\mathcal{U}$	mesh refinement operator

Chapter 1

Introduction

The Partial Differential Equation (PDE)

$$-\nabla \cdot \epsilon \nabla u = f \tag{1.1}$$

on well-shaped domain Ω is a building block within many applications [4]. We are interested in solutions $u : \Omega \mapsto \mathbb{R}$, with $f : \Omega \mapsto \mathbb{R}$ and $\epsilon : \Omega \mapsto \mathbb{R}^+$. This equation is the Poisson equation when $\epsilon = 1$. The Poisson equation, and its generalised variants, have many practical applications, such as dispersal of oxygen in tissue [5], or water saturation in soil [6], [7]. They model diffusion. Poisson is not only common as a problem in and of itself, but also as a subproblem within a larger system, such as determining the friction in both the compressible and incompressible Navier-Stokes equations [8], [9]. For problems with time stepping, the Poisson equation, or its cousins, must be solved for each and every time step [10], [11]. Repeated solves throughout the simulation put a great onus on quick time-to-solution.

Due to the highly variable nature of PDEs, different categories of PDE require different approaches in order to be solved efficiently. There is no one hat fits all solution. As we are interested in diffusion, we look to elliptic PDE solvers—in particular multigrid [12], [13]. Multigrid solvers are known to be among the best in the world for solving linear elliptic problems, but have also been shown to be effective for certain classes of non-elliptic and nonlinear problems. They have been widely applied in

many existing systems, and though multigrid solvers are a relatively mature family of solvers—they converge in an optimal number of compute steps for certain problems and show convergence rates that are independent of the problem size—there still remain many challenges. Our aim is overcome some of these challenges. A multigrid solve can be split into two broad phases:

1. The assembly phase.
2. The multigrid cycles themselves.

We reduce the overall time-to-solution of the solve by reducing the run-time of both of these components separately.

To improve both phases within the overall multigrid solve, we must take into account many factors that effect the overall run-time. Firstly, we improve the assembly phase. We reduce the time-to-solution by reducing the delay due to initial assembly cost; we introduce a lazy form of assembling the fine grid equation. Rather than assembly being a discrete (i.e. separate) phase that occurs prior to the start of the solve, we present it as a discreet (i.e. obscured) phase that occurs in parallel with the rest of solve. The solve starts before an accurate assembly process has terminated. Instead of an accurately assembled matrix, the solve uses an initial approximation that is cheaper to compute. Secondly, we improve the multigrid cycles themselves. We reduce the time-to-solution by introducing a method of stabilising additive multigrid. As we move along the path to exascale, improvements must be made to parallel multigrid solvers—additive multigrid is a multigrid solver that shows great potential for parallelisation, so is our chosen multigrid flavour. However, additive multigrid is less robust than its alternatives [14]. We improve the robustness via stabilisation parameters—specifically by introducing a *vertex*-specific damping parameter that is dependent upon the current solution value held in a specific vertex. This gives rise to a new category of additive solvers: adAFAC-x. The damping parameters improve stability without dramatically reducing the convergence rate.

We delay the integration of accurate stencils until the solve is well underway to

improve the assembly process. This phase has reduced concurrency compared to the later multigrid cycles due to limited arithmetic intensity. We overlap the solve phase with tasks from the assembly to increase parallel potential. The assembly phase is now implemented as a series of background tasks that can be deployed to cores that would otherwise be idle. This series of background tasks is an iterative sequence that will eventually return an accurately integrated fine grid stencil. In effect, we increase the workload performed by the assembly process, but we observe a reduction in time-to-solution compared to a priori assembly as we have increased the possibility for concurrency.

We introduce a new class of solvers: adAFAC-x. These are *a*dditively *d*amped *A*synchronous *F*ast *A*daptive *C*omposite grid solvers, to improve the solve phase. This is a category of solvers that constructs damping parameters on an auxiliary coarse grid totally asynchronously to the main solve. They are inspired by the fast adaptive composite grid (FAC) family of solvers, in particular AFACx [15]–[17]. Both AFACx and our adAFAC-x solvers belong to the class of additive multigrid solvers. Additive multigrid solvers allow for more concurrency than other multigrid solvers, as they are not required to process coarse grids in a fixed order. Stability issues are seen when a large number of grid levels are used or when solving more challenging underlying equations—problems with large discontinuities in a material parameter ϵ for example. These limit the classes of problems additive multigrid can effectively be applied to as well as the practical sizes it can scale to. Existing methods to recover stability are expensive or limit convergence rates. Our adAFAC-x solvers have proven particularly effective for problems that have large discontinuities in the material parameter and scale over a greater number of coarse grid levels than baseline additive multigrid. They are also suitable for dynamically adaptive meshes.

When implementing both of our novel ideas, there are two trends in modern hardware development that we focus on directly incorporating into algorithmic design. These are: a movement towards massively parallel systems with a large number of cores spread over a large number of compute nodes; and a widening gap between processor

clock speeds and memory access speeds [18]. To fully take advantage of a large scale high performance computer, an implementation needs to fully utilise all available cores (which may be in the hundreds, if not thousands), while not falling victim to the limited bandwidth for communication between compute nodes. A solve must scale all the way to exascale. An algorithm may require minimal compute steps, but if the run-time is dominated by communication then it cannot scale. Performance for a single node can also be dominated by periods of non-computation. This is due to increasing memory access times—loading data from main memory onto the chip, when the data is only required for a small number of computations, can kill performance. If neither of these features are incorporated into the algorithm's design then an implementation will be unable to scale. Through this we hope to address some fundamental issues that prevent scaling to exascale. We have eliminated synchronisation points in existing approaches and found additional ways to exploit hardware concurrency. This means we do not solve problems that are exascale, rather we address some **algorithmic** problems with existing exascale approaches. We investigate new ideas that could be used by others when solving exascale sized problems.

The structure of the thesis is as follows: We open with a brief overview and summary of our work (Chapter 2). Then in Chapter 3 we provide a detailed explanation of our motivation, cover related work, and introduce existing ideas that we build upon. The following three chapters explain in detail our contributions to the state of the art. These are split into three categories: Firstly, our asynchronous assembly of operators to mitigate setup costs (Chapter 4); secondly, our method of stabilising additive multigrid using vertex-specific damping (Chapter 5); and finally, our implementation of both of these concepts in a single-touch variant (Chapter 6). In Chapter 7 we showcase a selection of results to examine the efficacy of our ideas. We close with a discussion of our work and an examination of possible future work in Chapter 8.

Chapter 2

In a Nutshell

The following chapter is a quick summary of our key contributions. We give a brief and informal overview of our main ideas and highlight our key contributions—these are our adAFAC-x solver suite and use of asynchronous stencil computation. This provides some intuition to our approach and lays the groundwork for the detail of our ideas. Subsequent chapters explain the concepts in detail, but this initial summary explains the "big picture" to illustrate how individual components fit together.

Multigrid solvers are generally viewed to be among the best solvers in the world [12]. They come in two main varieties: additive and multiplicative [14]. Both flavours are multilevel correction schemes. An error approximation—the residual—is computed on the finest grid level which is restricted to a set of coarse grid levels; for additive multigrid specifically updates can then be computed for all grid levels in a single grid traversal. The multiplicative variant of multigrid—where the grid levels are run through sequentially—is known to yield optimal convergence rates in many scenarios. Additive multigrid, in contrast, does not share this feature due to overcorrections: multiple corrections across grid levels all push the solution too far in the same direction causing the solution value to "overshoot". Therefore the performance of additive multigrid is often worse than multiplicative. On the face of it, this implies that additive multigrid is the lesser of the two options and only useful as a preconditioner. However, damping corrections—i.e. scaling them—within an additive scheme pre-

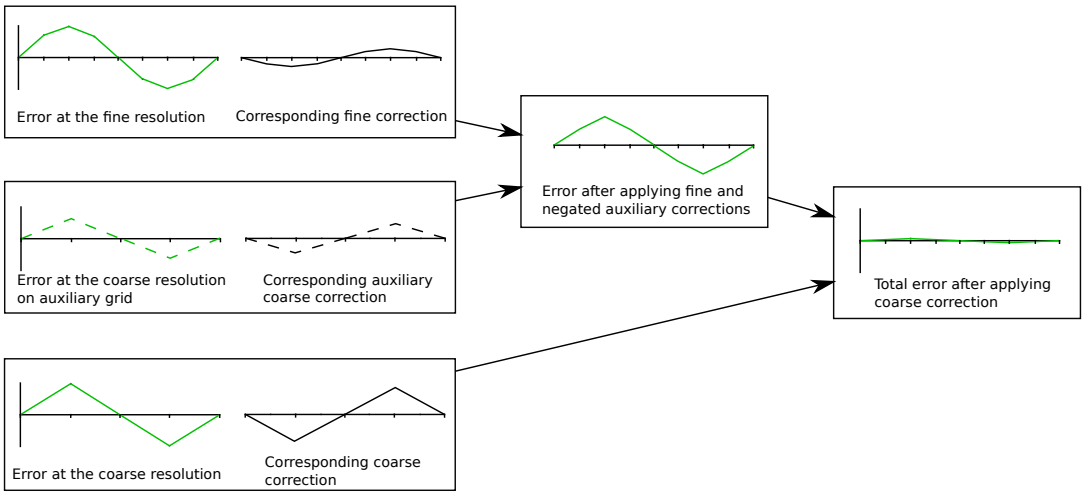

Figure 2.1: Summing of corrections within adAFAC-Jac. We damp fine grid corrections with corrections from an auxiliary grid—we now only remove resolution specific errors. This prevents overshooting when we introduce coarse corrections.

vents these overcorrections, making additive multigrid more stable as a solver. Level-specific damping, for example, is an effective method of removing the overcorrections, although it harms the overall rate of convergence. Uniformly reducing the impact of corrections across levels recovers stability, but harms the rate of convergence—long distance correction contributions have their impact removed. Damping in this fashion is in opposition to a fundamental feature of multigrid that make it so effective—that updates propagate throughout the entire domain. **We therefore develop a new scheme—adAFAC-Jac (*a*dditive *d*amped *a*sychronous *f*ast *a*daptive *c*omposite grids with *Jac*obi smoothing)—a damping scheme that avoids this limitation**. Stabilising additive multigrid makes it more competitive as a solver, so it can be used not purely as a preconditioner. Additive multigrid has better parallelisation properties than multiplicative multigrid, which are become incredibly desirable in the exascale era.

Our novel damping term tries to mirror the component in coarse grid corrections that is "too much"—we want to exclusively remove the part that leads to overcorrecting. This goes beyond a simple parameter computed per-level—instead, we compute a

different damping term per-vertex. Multiplicative multigrid does not exhibit such overcorrection, so we assume overcorrections can be negated by computing a damping term that approximates the difference between an additive multigrid update and a multiplicative one. The damping term imitates the missing multiplicative impact, therefore we subtract it from the additive correction. This damping parameter is computed additively, so does not reduce potential parallel performance. We introduce an additional—"auxiliary"—correction space for each grid level within the multigrid hierarchy: each space produces extra corrections that "correct" the original corrections. The residual on each original grid level is additionally restricted to the auxiliary correction spaces; however, this secondary restriction uses a partially "smoothed" restriction operator. For a matrix A, smoother M^{-1}, and existing restriction operator R we restrict to the auxiliary grid space using the matrix triple product RAM^{-1} (which we can precompute/hard-code). Our chosen smoother is a Jacobi smoother, so M is simply the diagonal of A. Each auxiliary space exists "on top" of an existing correction space: It covers the same topological region as the original grid space, but with increased mesh spacing. The increased mesh spacing means auxiliary correction spaces match to existing coarse grid correction spaces with the same mesh spacing, i.e. the existing coarse grid space one level coarser within the mesh hierarchy. When restricting to the auxiliary coarse grid space, the partially smoothed restriction operator accounts for components within the residual that are eliminated by fine grid smoothing. Once damped, the contributions from the original grid spaces have effectively isolated the errors that exist exclusively on that resolution. An illustration of how we target different error components is seen in Fig. 2.1.

The solver algorithm is not the sole determining factor for time-to-solution when solving a PDE. Before a solve can start, the grid and matrix must first be initialised which can become expensive. It can become particularly costly if we use a regular grid that gives a desired (low) discretisation error, due to the sheer number of grid points required. Additional grid points increase both the memory cost, and the

Figure 2.2: An example of a possible material parameter within a cell (left). A three stage splitting of the cell into regular grids of subcells, to capture the material data with increasing accuracy. The subcells have resolutions of 2x2, 8x8, and 64x64 (right).

number of computations that must be performed, thus delaying the solve start. **We propose to re-order operations such that the solver algorithm starts earlier in the pipeline, i.e. to not wait for exact matrix equations to be computed. Instead, the solver will start using less accurate stencils.** More accurate stencils are then iteratively computed in parallel with the solver steps. We approach this in two ways: in how we handle the coarse grid equations, and in how we handle the fine grid equations.

Rather than computing the fine grid equations to a high level of accuracy prior to the first steps of a solve, an inaccurate initial approximation is instead used. This is a similar principle to adaptive mesh refinement (AMR), where a simple grid is used to kickstart the solve. In an AMR scheme, additional vertices are added as the solve progresses, so the grid develops alongside the solution itself [19], [20]. Within our scheme, the early iterations solve an inexact matrix equation—we assume the solution to this equation is similar to the true solution, so the solution approximation is still pushed towards the true solution. Accurate equations are computed in parallel to the solve. These updated integrations are performed using cores that would otherwise be idle if the solver exhibits low levels of concurrency—this additional level of parallelisation means we see a reduction in time-to-solution relative to the conventional method of equation assembly. The solve then sees no delay on when it is able to progress due to the computation of fine grid equations.

A sample splitting of a single cell into subcells to capture material data can be seen in Fig. 2.2. Here the material parameter is sampled in the centre of the subcell and treated as constant over the subcell. Fine detail representation of the material parameter is only seen when using a large number of subcells.

This idea carries through to the coarse grid equations. We use an iterative procedure to construct both fine and coarse grid equations. Subsequent fine grid stencil iterations increase the number of material parameter points incorporated to improve the accuracy, so coarse grid equations are iteratively (re-)computed using the currently held fine grid equation. Updates on the fine grid ripple up through the coarse grid levels one level at a time. This is particularly effective when AMR is used, as each refinement changes the grid, so the entire equation hierarchy also changes. We assume matrix equations change minimally after a refinement. Therefore, the new matrix is not dramatically different from the old. Rather than delaying the continuation of solve with an equation recomputation step, we delay the computation of the equations instead, so the solve continues with the old equations. We target scenarios where on-the-fly rediscretisation is insufficient—our primary use case is the Poisson equation with non-trivial variable coefficients. Here, rediscretisation and simple geometric interpretations are only conditionally stable or lack robustness [12], [21], [22]. Algebraic construction of coarse grids therefore incurs substantial additional assembly costs on every grid refinement. We avoid this with our use of approximate coarse grid equations.

We integrate all these components into a single solver. To the best of our knowledge, this is the first asynchronously constructed, additive multigrid scheme that implements locally adaptive damping. Our solver works on fully adaptive grids, is stable under a wide range of conditions, and takes advantage of a single-touch policy. We pipeline memory accesses, so that, in general, most memory locations are written to, and read from, only once per grid traversal—this can be said to be an optimal total number of read/write operations.

Chapter 3

Motivation and Related Work

There exist many challenges for solvers that we must first understand if we hope to successfully address them. Here, we briefly outline specific challenges existing solvers face and give an idea of what current approaches attempt. Further detail is given to approaches that we build upon and specific issues within those approaches. We split challenges into three broad categories: the first is the efficiency of the setup; the second is the efficiency of the solver; the third is the implementation fully taking advantage of underlying hardware. For the first two groups, efficiency specifically refers to algorithmic and mathematical efficiency, and by extension robustness. A non-robust solver can be viewed as an inefficient solver—solvers will switch to a more robust, and likely more expensive, approach if their current approach is not converging—therefore improving robustness also improves efficiency.

The following chapter includes sections modified from text that was previously published in [1], [3]. The Sections 3.2.3, 3.2.5, 3.2.7, as well as parts of 3.2.6 and 3.3.3 are expanded from "Stabilised Asynchronous Fast Adaptive Composite Multigrid using Additive Damping". Section 3.3.2 and other parts of 3.3.3 are expanded from "Delayed approximate matrix assembly in multigrid with dynamic precisions".

3.1 Assembly

3.1.1 Mathematical framework

We start with the mathematical fundamentals: In order to solve an equation numerically, we must first represent this equation within a computer. This overall process is the assembly process and is a phase we seek to optimise. We start by introducing the mathematical basics and outline some existing methods of representing equations discretely/numerically, more specifically, we introduce the Finite Element Method [23]. Our work either builds on ideas covered here or uses lessons learned from the methods to inform the development of our ideas.

Definition 3.1.1. Assembly: The holistic process of creating data structures, allocating memory, and filling said data structure with the correct information.

We are numerically solving the partial differential equation (PDE) $-\nabla \cdot (\epsilon \nabla u) = f$ on the domain Ω with Dirichlet boundary conditions. That is, we are looking for an approximation of the function $u : \Omega \mapsto \mathbb{R}$, with $\epsilon : \Omega \mapsto \mathbb{R}^+$ as a material parameter and $f : \Omega \mapsto \mathbb{R}$ as the right-hand side. The entire equation must be discretised in order to be solved—our chosen method is a finite element method. For finite elements we start by rewriting the PDE in its weak formulation. This rewrite involves integrating against an arbitrary test function $v \in V$. We take V as a space of functions with local support and restrict $v = 0$ on $\partial\Omega$. This gives the weak formulation of the PDE as

$$\int_\Omega \epsilon \nabla u \cdot \nabla v dx = \int_\Omega f v \, dx, \tag{3.1.1}$$

where u is represented as a sum of weighted basis functions $\sum_{i=0}^{n} u_i v_i$, u_i is a vertex weight and v_i a basis function. Basis functions are selected from the same space as the test functions, i.e. we use a Ritz-Galerkin formulation. We work with scalar equations and our nodal basis is chosen to be the vertices of a Cartesian mesh with d-linear basis functions. Throughout the text, we therefore use the terms vertex, degree

of freedom and weight function interchangeably. Each test function corresponds to an integration over this region of local support and creates a linear constraint on the weighted basis functions that exist within this region of support. That is, (3.1.1) can be decomposed into a series of integrations over individual elements

$$\int_{\Omega_h} \epsilon \left(\nabla u, \nabla \phi\right) \; dx = \sum_{c \in \Omega_h} \int_c \epsilon \left(\nabla u, \nabla \phi\right) \; dx. \tag{3.1.2}$$

The integrations therefore define a system of linear equations, i.e. a (sparse) matrix. Solving this approximation to the PDE then becomes a matter of inverting this matrix [24].

Although we require a representation with high accuracy, this does not always give rise to an obvious choice. Different bases give different quality representations for different PDEs. For finite element discretisations, users generally commit to a single class of basis functions, such as d-linear basis functions, with which they discretise the domain spatially. This automatically yields a function space discretisation. As the PDEs we are specifically interested in solving are those with non-trivial material parameters, we focus on how methods represent such parameters in discretisations. Careful positioning of the nodes/elements within a finite element computation can exactly capture such features [25], [26].

However, simply adding additional degrees of freedom to resolve features is not an ideal solution. Additional degrees of freedom increase the computational cost of assembly and of the solve. Therefore, high discretisation accuracy for a fixed, or low, number of elements is a priority. We want to keep assembly time low. The freedom to move the position of nodes, or even change the order of basis functions, is a logical choice due to these factors. Adaptive finite elements [27], [28] is one example of a scheme that changes the order of shape functions and nodal position to craft a more accurate equation. However, higher order schemes may be less stable than lower order schemes and thus require a solver to fall back to a lower order scheme—such as switching from a high order discontinuous Galerkin scheme to low order finite volume limiter when the local solution violates predefined physical admissibility

criteria [29]— to improve convergence rates. With this in mind, we focus on low order schemes and aim to minimise the assembly time.

In general we are required to perform a numerical integration to assemble the matrix—therefore it is this numerical integration we must optimise. Complicated/non-trivial features within a PDE mean assembling the matrix analytically becomes costly or impossible. Numerical integration of the PDE becomes expensive as arbitrarily complicated features—e.g. material parameters—render the computation non-trivial. Methods such as the Finite Cell Method [30], [31] develop efficient integration quadrature rules using subcell splitting of elements. Adaptive quadrature schemes, those schemes with dynamic termination criteria, are very cost-effective. They do not fix the discretisation a priori and instead perform initial evaluations and use error estimates to determine termination criteria. They are harder to implement than static discretisations and the overall cost of assembly is unknown prior to the start of the assembly process. Our own work uses adaptive integration schemes within the assembly process.

Overhead from the data structure management in the assembly process also adds to the cost. Without a careful choice of basis functions enforcing a specific sparsity pattern for the matrix (such as the basis functions having local support), the matrix itself can be dense. Many solvers involve the application of the original matrix to a solution vector—applying a dense matrix is expensive. Dense matrices destroy data locality patterns and therefore require additional memory access time. The additional memory overhead inherent to storing the matrix is a limiting factor for scaling to large problems and systems [32], [33]. Data movement also limits scalability. We focus on equations with complex material data, this material data must be streamed onto the chip. A large amount of data is thus moved to compute a proportionally small number of scalar values. Such movement strains the memory interconnect when loaded and delays computations, reducing arithmetic intensity [34].

An existing method of negating the initial discretisation phase is a matrix-free implementation. Switching to a matrix-free scheme reduces the memory requirements

of a solve, but does not reduce the memory access time. Such an implementation does not have an initial assembly phase prior to the start of the solve and does not explicitly store the matrix. Instead, local stencils are hard-coded, or assembled, as and when they're needed. There is no persistent storage of the matrix. This is effective for solvers that only use geometric information about the PDE and thus do not require additional data to be streamed onto the chip. It is less effective for solvers that require additional processing of features, such as material data—multigrid solvers that construct algebraic coarse grids and require information about the structure of patches of the matrix, for example.

Our chosen method of discretisation is finite elements. They are a flexible discretisation scheme that is effective in the presence of large discontinuities and stencils can be assembled in parallel.

3.1.2 Meshes and adaptivity

With a discretisation scheme at hand we now must decide on the most appropriate method of storing this information in memory. Different choices work better for different solvers—we are most interested in an effective choice for an adaptive multigrid scheme. An adaptive mesh amortises the assembly cost, and is an idea we follow up on in our work. We briefly summarise a few existing options for discretisation and their properties herein.

As we implement a multigrid solver, we work in the general realm of iterative schemes. Any iterative scheme can be viewed as a matrix-vector product (mat-vec). Multigrid itself involves the repeated application of an approximate inverse of the matrix (a smoother) to a residual vector, and Krylov schemes involve repeated application of the matrix to construct Krylov subspaces. For a stationary iterative method, inverting the matrix becomes a process of repeated application of a matrix to a solution vector. Within multigrid specifically, multiple mat-vecs are generally required for residual updates and the smoother application. Therefore reducing the

memory requirements for the matrix and memory access time of matrix elements are two key concerns. In order to apply the matrix to a vector, a data structure to store the matrix and the solution must both be chosen. Naively storing the matrix as a series of rows and columns requires a large amount of memory and necessitates indirect memory accesses. It is simply not feasible. A possible, smarter, implementational solution would be a matrix storage format such as Compressed Row Storage. However, we approach the problem along a different tack.

Encoding the matrix onto a mesh is our chosen solution. A structured grid is a simple example of a mesh. In a structured grid the connectivity between vertices, and by extension the layout of the entire mesh, is fixed. The same local structure is repeated throughout the mesh. Structured grids, particularly regular grids, offer many advantages. The simple structure allows matrix equation construction and implementation to be similarly simple. Memory access patterns are defined before the solve starts, limiting cache stalls and reducing latency, due to data being fetched from main memory [35]. However, as meshes become increasingly fine, more mesh points are required. Sufficiently fine regular grids can therefore require a prohibitively large number of vertices, which increases both memory requirements, and compute time for each iteration in an iterative solve, due to processing time for additional vertices.

Structured grids are simple to implement but may not be the optimum grid for all equations. A changing material parameter requires careful placement of vertices. Meshes are not limited to being structured—instead vertices can be placed totally arbitrarily with varying connectivity, this reduces the discretisation error for a fixed total vertex count and more accurately represents material parameters. Features within the material parameter can be totally represented within the mesh itself with careful positioning of the vertices, as seen in Fig. 3.2, which reduces discretisation error. Unfortunately, due to the fact that mesh cells can be arbitrary shapes, the condition number of the resultant matrix can become arbitrarily large.

Smaller meshes are better than larger meshes for one key reason: A small mesh

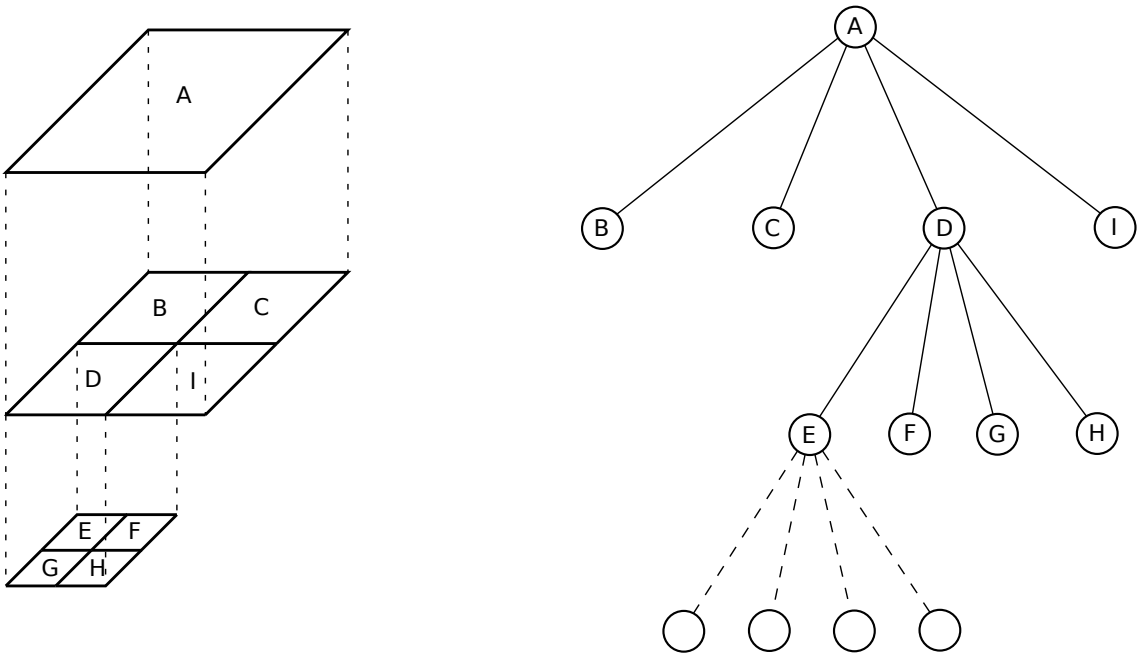

Figure 3.1: An illustration of a Cartesian mesh as a hierarchy of regular grids (left) and a mapping of the same cells directly to a tree (right).

means small matrix. Adaptive Mesh Refinement (AMR) can reduce the number of points in a mesh, while still capturing features of the solution or equation directly in the mesh, this was developed in seminal work by Berger and Colella [36], [37]. In AMR, the mesh, and thus specific discretisation, is not fixed. For dynamic AMR, the mesh changes through the solve, whereas, for static AMR, the mesh is fixed once the solve itself starts. For either approach, we start with an initial guess of the mesh and add additional nodes in regions of interest. The work by Berger and Colella is specifically for hyperbolic equations—we use similar ideas, although we focus on elliptic equations, so concepts do not map one-to-one. In early work on AMR for elliptic equations, Achi Brandt stated "discretization and solution processes are intermixed with, and greatly benefit from, each other" [19]. The goal of AMR is to reduce the number of degrees of freedom within the discretisation, which results in a final mesh with high accuracy fine detail representation only in regions where it is most required. However, a feature of AMR, that we exploit, is an avoidance of an initial costly assembly phase. Such phases delay the start of solve. The mesh instead unfolds as the solution develops. The assembly cost is not negated, in fact it may

be increased, but it is amortised over the duration of the solve. After a refinement, a re-assembly process must be performed when setting up the new mesh—repeated re-assembly processes massively increase the total cost of assembly. However, if the equations are known a priori, e.g. Laplace equation on a Cartesian mesh, then this delay is not seen.

This is the general theory behind AMR but says nothing about the implementation and data structures. An adaptive mesh can, rather intuitively, be achieved with an unstructured mesh. Whilst iterating towards a solution, additional vertices are added in regions where they reduce the discretisation error the most. The lack of structure means there are less strict constraints on the placement of additional vertices. The connectivity of the mesh, and by extension the fine grid equation, change with the addition of each new vertex. This triggers costly, and possibly non-local, re-assembly phases. Similarly to general unstructured meshes, new vertices can increase the condition number of the matrix. This, coupled with the fact that unstructured mesh assembly is more complicated than assembly for structured meshes, results in a reassembly process that quickly becomes painful.

Structured meshes can give rise to cheaper assembly phases than unstructured meshes and associated matrices have additional structure that further benefits solvers. Combining the best elements of AMR and structured meshes is therefore not unheard of. An adaptive mesh can arise from the composition of multiple structured grids [19], [38]: an initial coarse regular mesh is constructed, then additional (finer and regular) grid spaces are introduced in regions where fine detail representation is of greatest benefit. Composite grids are one implementational solution—the overall grid being a composition of multiple separate subgrids. The subgrids can be arbitrarily placed and overlap; many different subgrids can exist in the same region of space if they correspond to different grid levels. Different subgrids of the same mesh resolution can exist and also need not be connected—the key constraint is meshes of the same resolution must not overlap. The composition of all subgrids (including the original coarse grid) is the effective grid. Locally a grid is regular; however, the

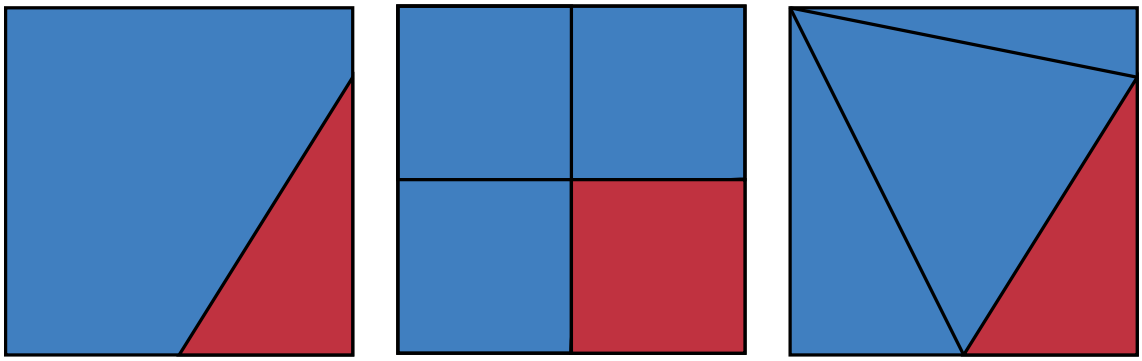

Figure 3.2: Attempts to accurately capture a material parameter (left) using only four cells for both a Cartesian mesh (centre) and irregular mesh (right). This geometry can be accurately represented on the irregular mesh but not the regular mesh.

mesh resolution is not constant across the domain. A subset of the structured AMR approach is to use a space-tree [39]–[42]. A space-tree can be viewed as a degenerated block structured mesh—the embedded fine grid patches are child elements in the tree. A simple diagrammatic representation of a space-tree can be see in Fig. 3.1. Quadtrees (and their higher dimensional analogue octrees) recursively split the domain into smaller and smaller cells, and embed these directly into the coarser cells [43]. Space-trees are widespread and have many different notations and implementations: we specifically follow on from Weinzierl's work on the Peano framework [42], [44] and use that implementation as a baseline.

We chose to work with structured grids due to the relative simplicity of implementation. A simple grid gives a simple baseline for assembly and numerical integrations therein—a structured grid produces a more readily parallelisable workload for the assembly. Adaptive grids, specifically space-trees, mean we are still able to take advantage of AMR.

3.1.3 Multilevel assembly

All of our novel ideas are for multigrid—one aspect of multigrid we contribute towards is the construction of the required matrices. We briefly cover what matrices must be constructed in a multigrid assembly phase and some options for the choice of these

matrices and their assembly. We are particularly interested in approximate methods of assembly for multigrid matrices.

Multigrid does not just require a single matrix equation and grid level to be constructed. It requires a cascade of matrices and grids [12], [13], [45]. Every grid level in the multigrid hierarchy requires a defined set of grid points. Every grid level in the multigrid hierarchy requires a matrix equation. Every grid level in the multigrid hierarchy requires its own assembly process. Similarly to the construction of the fine grid, and the resultant equation, the construction of coarse grids can also be expensive. Coarse grid assembly can be split into three components: the method of coarsening fine grid vertices to define the coarse grid, the construction of the coarse grid equation, and the identification of intergrid transfer operators. We now cover possible choices for these, as the specific choice drastically effects how effective a multigrid solver can be.

Any multigrid implementation will also make use of a smoother. A smoother requires two components to be assembled–the matrix being solved and an approximation of the inverse of the matrix. The original matrix is involved in the residual calculation—strictly speaking this does not need to be fully assembled, but information about the matrix is required for an accurate residual to be computed. The residual calculation is a mat-vec product. In our work, we mitigate certain issues in the assembly, by enforcing a specific sparsity pattern due to our choice of basis functions (d-linear basis functions on a Cartesian mesh) so the matrix is sparse and shows good data locality. We also split stencils into element-wise decompositions, which further maintains data locality. The approximation of the inverse also does not have to require a global assembly procedure. We perform Jacobi smoothing, ergo we only require knowledge about the diagonal elements of the matrix. Jacobi smoothing formally requires multiple mat-vec products, but, in practice, we can implement the application of the diagonal matrix as a scalar product between the extracted scalar diagonal element and an already computed scalar residual to eliminate one mat-vec. The mat-vec to compute the residual remains.

Possible coarsening strategies for multigrid fall into one of two approaches: algebraic multigrid (AMG) and geometric multigrid (GMG). AMG directly uses the matrix to define the coarse grid [46], [47]. That is, no geometric information from the fine grid enters the coarse grid. Instead, the fine grid equation, and the strength of coupling between fine grid points, is used to identify a set of vertices to constitute the coarse grid. This is not a cheap process. GMG, on the other hand, uses a much less expensive method of coarsening. The coarse grid vertices are defined exclusively using geometric information—this could be as simple as selecting alternate fine grid points for a one-dimensional mesh. The coarse mesh can therefore be defined a priori. As we use structured grids, we use a geometric coarsening strategy.

Coarse grid equations can also be defined via one of two approaches: through rediscretisation or via an algebraic definition. In our work we incorporate both as different options. For rediscretisation, coarse grid error equations are defined by simply discretising again on that grid level [12]. This is relatively quick to compute and can be constructed at run-time to avoid explicitly storing and constructing the matrix. Rediscretisation can have limited effectiveness, however, as solvers can exhibit instabilities [35], [40]. Alternatively, there are algebraic approaches. The standard algebraic approach for coarse grid equations is Ritz-Galerkin, i.e. the coarse grid equation A_H is defined as the result of the matrix triple product $A_H = RA_hP$. Ritz-Galerkin is the defacto method for AMG: the coarse grid does not use any geometric information, so must be defined in terms of the fine grid equation and intergrid transfer operators. The intergrid transfer operators, R and P, should be constructed in a way so that coarse corrections do not introduce an error on the fine grid. Algebraic methods may define R and P algebraically, or they may stick to geometric definitions, e.g. d-linear interpolation. For certain grids, rediscretisation and Ritz-Galerkin with geometric operators may give the same coarse grids, e.g. for Finite Elements on Cartesian meshes. Algebraic approaches are generally more flexible as a solver and as such can be applied to a wider range of problems than geometric multigrid [12], [48], [49]. However, they have a costly setup phase—the

coarse grid identification and coarse grid equations construction are expensive [50], [51]. Some geometric implementations are shown to also share costly coarse grid assembly phases [52].

The complex material parameters we focus on can result in complex solution behaviour, as relevant features might exist across multiple scales. When data is multiscale, we must also incorporate this into the coarse grid representation. Homogenisation is a general form of this [53], [54]. Material parameters can be too fine to accurately be represented on even the finest computational grid, so coarse representations of the material parameter must be "averaged" over the cells. The material on the macro scale then has the same effective property as the true material parameter. Harmonic, arithmetic and geometric averaging are simple implementations with limited effectiveness [55]. More advanced approaches, such as renormalisation or methods that separate the diffusion coefficient from the equations, have seen greater success [56], [57]. The Ritz-Galerkin coarse grid formulation can be viewed as a form of homogenisation [58] and is a standard element in the multigrid toolkit. This idea is built on with algebraic multigrid. All of these accurately capture fine grid information on the coarse grid but either suffer from increased assembly time or a less effective solver and increased solve time.

A common application of multigrid is as a preconditioner. Preconditioners are a wide class of approximate solvers that do not bother themselves with inverting an equation exactly. They instead push the solution in the correct direction while another solver handles the details. An approximate preconditioner for multiscale problems is the appropriately named Multilevel Sparse Approximate Inverse preconditioner [59]. Here, there is a focus on AMR: Changes in the mesh require global changes in the matrix—this preconditioner, however, does not require exact matrix inversions to be effective so instead they use an approximation that is only depends upon local mesh information. We posit that in an iterative solver, the idea behind a preconditioner—not solving the exact equation but a similar equation that will push the solution in the correct direction—is one that can be repurposed.

3.2 Solver ingredients

3.2.1 Solution characteristics

To construct an effective solver for an equation we must understand the behaviour and general character of solutions and the underlying problem. We develop a parallel solver for elliptic equations that is designed to be able to scale to exascale, so are interested in existing examples of this approach.

Elliptic operators are particularly challenging to solve. The Poisson equation is the simplest example of an elliptic operator [23], [24]. A consequence of ellipticity is that local changes effect the entire domain. The discretisation can be poor in two ways: Firstly, poor spatial discretisations lower solution quality and accuracy. This occurs when there is no local refinement around a singularity, for example. They can cause cause pollution [60]. Secondly, poor operator discretisation, which can take two forms—poor discretisation on the fine level and the coarse level. A low quality fine grid operator integration creates an inconsistent discretisation—the solver is solving a different problem than intended. A low quality fine grid operator integration instead causes another type of pollution [32]. Coarse grid updates will not fall into the nullspace of the fine grid operator, i.e. coarse corrections introduce error after projection. This worsens the rate of convergence.

Poor integration becomes an issue when material data is not constant. A discontinuity in the material data, for example, effectively decouples the two regions. A non-continuous material parameter [21], [61], may induce jumps in the first derivative of the solution. This causes certain schemes, such as finite differences, to suffer. Other schemes, such as finite elements, handle discontinuities better when the integration is performed accurately. A poor fine grid discretisation pollutes the solution as the discretisation does not accurately resolve these features. Fine grid detail may not be accurately resolved on the coarse grids. A poor coarse grid discretisation smears over the discontinuity and introduces errors. Both factors cause additional

oscillations around a material parameter discontinuity when converging towards a solution.

Larger jumps of the material parameter over a discontinuity cause a matrix representation of the system to have a worse condition number. Higher condition numbers correspond to worsened rates of convergence for a solver—the number of steps an iterative algorithm requires to converge increases. For a multilevel solver, e.g. multigrid, this leads to large oscillations over the discontinuity, and sufficiently large jumps in the material parameter causes such codes to diverge [21], [61]–[64]. Damping out the oscillations in this region increases the stability of a solver. A discontinuity is a highly localised anisotropic region along the jump itself, ergo a large magnitude difference in the material parameter effectively decouples the regions, and harms the convergence rate.

3.2.2 Additive multigrid

The specific flavour of multigrid we are interested in is additive multigrid. Our choice of a solver is an additive solver which is known to suffer from stability issues. We briefly explain the basics of additive multigrid and why it is often unstable.

One formulation of the additive multigrid method is given by the following algorithm:

Algorithm 1 Additive algorithm

function ADDITIVEMG(ℓ, u_ℓ, f_ℓ)
  if $\ell = 0$ **then**
    $u_\ell \leftarrow A_\ell^{-1} f_\ell$
  else
    $r_\ell \leftarrow f_\ell - A_\ell u_\ell$ ▷ Can be omitted: $u_\ell = 0$ means $r_\ell = f_\ell$
    $u_{\ell-1} \leftarrow 0$
    ADDITIVEMG($\ell - 1$,$u_{\ell-1}$,$R_{\ell+1}^{\ell} r_\ell$)
    $u_\ell \leftarrow u_\ell + M_\ell^{-1} r_\ell$
    $u_\ell \leftarrow u_\ell + P_\ell^{\ell+1} u_{\ell-1}$
  end if
end function

u_ℓ	Solution on level ℓ
b_ℓ	Right-hand side on level ℓ
A_ℓ	Matrix on level ℓ
M_ℓ^{-1}	Smoother (cheap approximation of A_ℓ) on level ℓ
$R_{\ell+1}^{\ell}$	Restriction from fine grid $\ell+1$ to level ℓ
$P_{\ell}^{\ell+1}$	Prolongation from level ℓ to fine grid $\ell+1$

The alternative to additive multigrid—the multiplicative formulation—smooths locally held values prior to the residual being restricted to the coarse grid [14]. Therefore coarse solves do not act upon the same residual. The algorithm for the multiplicative method is given in Alg. 2. The pre-smoothing steps, shown in blue, must be performed prior to the coarse smoothing, and the post-smoothing steps, shown in red, must be performed prior to projecting updates to finer grid levels. This enforces the sequential handling of the grid levels.

Algorithm 2 Multiplicative algorithm

function MULTIPLICATIVEMG($\ell, u_\ell, f_\ell, \mu_{\text{pre}}, \mu_{\text{post}}$)
  if $\ell = 0$ **then**
    $u_\ell \leftarrow A_\ell^{-1} f_\ell$
  else
    for $i = 1$ to μ_{pre} **do**
      $r_\ell \leftarrow f_\ell - A_\ell u_\ell$
      $u_\ell \leftarrow u_\ell + M_\ell^{-1} r_\ell$
    end for
    $r_\ell \leftarrow f_\ell - A_\ell u_\ell$
    $u_{\ell-1} \leftarrow 0$
    MULTIPLICATIVEMG($\ell - 1, u_{\ell-1}, R_{\ell+1}^{\ell} r_\ell, \mu_{\text{pre}}, \mu_{\text{post}}$)
    $u_\ell \leftarrow u_\ell + P_\ell^{\ell+1} u_{\ell-1}$
    for $i = 1$ to μ_{post} **do**
      $r_\ell \leftarrow f_\ell - A_\ell u_\ell$
      $u_\ell \leftarrow u_\ell + M_\ell^{-1} r_\ell$
    end for
    end if
  end function

Additive multigrid, on the other hand increases the potential for parallelism [65], [66]. We can rewrite additive multigrid from the recursive formulation in Alg. 1 to the loop based formulation seen in Alg. 3.

Algorithm 3 Additive algorithm as a for loop

function ADDITIVEMG($\ell, u_{\ell_{\max}}, f_{\ell_{\max}}$)
  $r_{\ell_{\max}} \leftarrow f_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{\max}}$
  for all Level ℓ **do**
    $u_\ell \leftarrow 0$
    $u_\ell \leftarrow M_\ell^{-1} R_{\ell_{\max}}^{\ell} r_{\ell_{\max}}$
    $u_{\ell_{\max}} \leftarrow u_{\ell_{\max}} + P_\ell^{\ell_{\max}} u_\ell$
  end for
end function

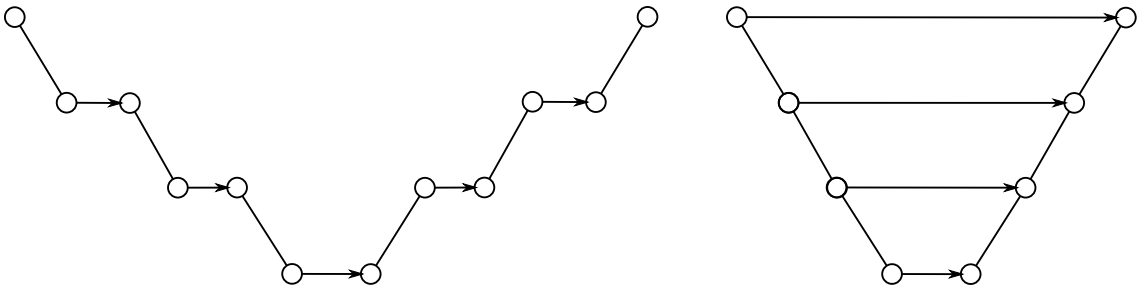

Figure 3.3: Representation of a multiplicative multigrid V-cycle (left) and an additive multigrid V-cycle (right). Horizontal arrows correspond to smoothing steps and the empty circles to accumulations/data transfers.

$\ell_{\max}$	Level index for the finest grid

For coarse grids, the residual computation shown is only there as a formality—u_ℓ is 0, so additive coarse grid residuals simply become the right-hand side of the equation. The same initial fine grid residual is restricted to the right-hand side of all coarse grid levels. An additive coarse grid level only requires information from the finest grid level. This is graphically shown in Fig. 3.3. Each loop can be handled independently, therefore additive multigrid does not need to smooth sequentially. The generalised matrix representation of additive multigrid can thus be written

$$u_{\ell_{\max}} \leftarrow u_{\ell_{\max}} + \left(\sum_{\ell=\ell_{\min}}^{\ell_{\max}} \omega_{\text{add}}(\ell) P_\ell^{\ell_{\max}} M_\ell^{-1} R_{\ell_{\max}}^{\ell} \right) (b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{\max}}) \,. \tag{3.2.1}$$

between data movement, V-cycles, for both variants is demonstrated As the same residual is restricted to all coarse grid levels, additive multigrid solves the same error equation across all grids—it is therefore prone to producing corrections to the same error on multiple grid levels [14], [32]. In multiplicative multigrid, however, local

errors are smoothed (read: removed) between transfers between grid levels. Thus, overcorrections are minimised. The multiplicative variant therefore converges in fewer iterations on a broader class of PDEs than the additive formulation.

The simple generalised matrix representation of additive multigrid is in stark contrast with that of multiplicative multigrid. As a point of comparison, we chose two minimal multiplicative multigrid variants, $V(1,0)$ and $V(0,1)$ cycles with exact solves on coarse grids. With factorisation, the $V(1,0)$ reads as

$$\begin{aligned} u_\ell \leftarrow & \left[u_\ell + \omega_\ell M_\ell^{-1}(b_\ell - A_\ell u_\ell)\right] \\ & + P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_\ell^{\ell-1}(I - \omega_\ell A_\ell M_\ell^{-1})(b_\ell - A_\ell u_\ell) \end{aligned}$$

and the $V(0,1)$ cycle as

$$\begin{aligned} u_\ell \leftarrow & \left[u_\ell + \omega_\ell M_\ell^{-1}(b_\ell - A_\ell u_\ell)\right] \\ & + (I - \omega_\ell M_\ell^{-1} A_\ell) P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_\ell^{\ell-1}(b_\ell - A_\ell u_\ell). \end{aligned}$$

Both multiplicative cycles require more matrix products and clearly do not share the level independence of additive multigrid.

Stability issues in additive multigrid worsen as the number of grid levels increases [22]. We therefore investigate methods of improving stability. The more grid levels there are, the greater the chance that overcorrections will be produced, which increases the magnitude of oscillations in the solution. Coarse grid contributions can be weighted with a damping parameter, ω, to reduce their impact on the fine grid solution and reduce the scale of overcorrections. Damping coarse grid contributions will improve stability but often at the expense of the rate of convergence. Currently there are few damping schemes that exclusively target the components within a correction that overcorrect [14].

With this in mind multiplicative multigrid may seem universally better than additive. However, the additional smoothing steps in multiplicative multigrid that improve the convergence rate also limit the maximum parallel potential [67]. A coarse grid update

cannot be computed until the fine grid has been smoothed and cannot be projected until all coarser grids have computed their updates. This creates a bottleneck. The coarsest grids are cheap by design, so most cores in a large scale machine will idle during the coarsest grid updates. Additive multigrid does not share this limitation. Updates across all grid levels can be computed in parallel as they share no data dependence [65], [66].

3.2.3 Improving intergrid transfer operators and BoxMG

Our end goal is to improve the stability of additive multigrid by reducing overcorrections; additive algorithms suffer from over correction between grid levels but in exchange gain increased potential for parallelism due to their potentially simultaneous handling of all levels. One approach to reduce overcorrections is to modify the intergrid transfer operators. We are interested in a couple methods of doing this, which we outline here.

An early example of this, which modifies the prolongation specifically, is by Greenbaum [64]. However, this does introduce a sequential element to the prolongation of corrections—coarse corrections are transformed so they are orthogonal to the updated fine grid residual vector once projected onto the fine grid. This reduces overcorrections compared to plain additive multigrid, but limits the potential for parallelism. Instead of modifying how corrections are projected, the restriction of the residuals could be modified instead. This is the approach taken by Chan and Tuminaro [68], [69]. They introduce a filter that separates the residual into two parts: one that exhibits errors with frequency on the resolution of the coarse grid; and one that exhibits errors on the fine grid resolution. Smoothers can therefore target errors specific to their mesh resolution. This reduces the amount of overcorrection/repeated correction between levels relative to conventional additive multigrid. However, this filter can be as expensive as the solve in and of itself.

Both the prolongation and restriction could be modified in the same solver. Due to

our focus on additive multigrid and our use of space-trees, we are most interested in methods of improving additive stability that also use geometric coarsening. Geometric coarsening defines the coarse grids using the topology of the fine grid and therefore require no complex computations to construct. Coarse grid equations can be constructed wholly using geometric information too; however, while this is simple to compute, it can limit the robustness of a multigrid solve. Therefore we look to existing methods that improve intergrid transfer operators to improve stability as this also improves coarse grid equations.

Dendy's Black Box Multigrid, BoxMG [21], keeps the simplicity of geometric coarsening, but constructs coarse grid equations that improve stability and rates of convergence. BoxMG operates on a Cartesian mesh and with an enforced sparsity pattern—the original outline is for two partitioning. BoxMG constructs algebraically inspired intergrid transfer operators, and corresponding coarse grid equations, for this coarsening scheme. The computed intergrid transfer operators should not introduce error on the fine grid when projecting coarse corrections, i.e corrections are projected into the nullspace of the fine grid. BoxMG has been extended to work on space-trees with three-partitioning [33], [70], [71]. Similar ideas of operator dependent transfers are explored by De Zeeuw [72] and a thicket of further work explores the interplay between AMG and GMG by using a forest of space-trees for GMG on the fine grids with an AMG scheme working on the coarse nodes in the forest [35], [40], [73].

To compute BoxMG intergrid transfer operators, all fine grid points are classified into either c-points (those points that coincide spatially with coarse grid points of the next coarser level), γ-points (points that coincide with the faces of cells of the next coarser level) and f-points (the remaining points that exist on the cell interiors of coarse grid cells). Weights must be computed that determine the impact of a coarse grid point when transferring to the fine grid (and vice versa). This can be seen in (Fig. 3.4). The weights are the entries of the matrix P.

- For c-points, prolongation and restriction weights are defined as the identity.

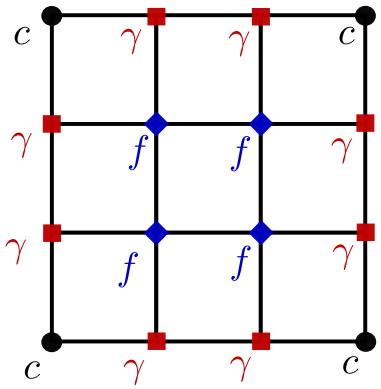

Figure 3.4: Splitting of fine grid vertices into c-points, γ-points and f-points for a BoxMG scheme on a space-tree that uses three partitioning.

- For points along coarse cell walls, i.e. γ-points, the fine grid stencil is "collapsed": If γ members reside on a face with normal n, the stencil is accumulated (lumped) along the n direction. The resultant collapsed stencil contains only entries along the non-n directions. Higher dimensional collapsing can be constructed iteratively. Applying this stencil to projected values from the coarse grid, is equivalent to applying the original stencil to coarse values that have been first projected along cell faces as normal, then linearly extrapolated in the n direction. The optimal prolongation along the cell faces is one where projected corrections do not introduce additional error, i.e. $APe = 0|_\gamma$. We solve a simplified version of this system: $\tilde{A}Pe = 0|_\gamma$. $\tilde{A}$ stems from these collapsed operators—for a chosen set of γ-points along a cell face and e is the characteristic vector for a vertex on the coarse grid, i.e. holds one entry 1 and zeroes everywhere else.
- Finally, weights for the f points, those on the cell interior must be computed. We can readily solve $APe = 0|_f$ for these remaining points. When computing these weights, f-points are only coupled to each other if they lie within the same cell's interior. This ensures the desired sparsity pattern for the transfer operator.

An alternative method of constructing the BoxMG coarse grid equations and intergrid transfer operators is explored by MacLachlan et al. [74]. Here BoxMG—and a

proposed adaptive variant—are compared to AMG and Adaptive AMG (Adaptive AMG [75], [76] is a modified AMG scheme that constructs the intergrid transfer operators iteratively). Tentative values are used for intergrid transfer operators initially with possible coarse grid candidates. Trial smoothing iterations are then applied to improve the transfer operator quality. Smoothing results therefore inform coarse grid definition. This ideology is applied to BoxMG to produce Adaptive BoxMG with relative success.

3.2.4 Modifying coarse equations, damping and BPX

Another method of improving the stability of additive multigrid is to modify the coarse grid equations being solved. This is an alternative approach we could incorporate into our solve. A popular form of modifying coarse equation is to damp them—which is a key part of our approach for improving stability.

Exponential damping is an aggressive example of damping. Grid levels are damped more aggressively the coarser they are [32], which is very effective at improving stability, but at the significant expense of convergence speed. A similar approach can be seen in Bastian and Hackbusch [14], where the same damping parameter is used in the smoother across all grid levels, but an additional damping parameter is encoded into the prolongation operator. The more levels a correction is projected across the more it is damped. Another approach, again mentioned by Bastian and Hackbusch, is to compute vertex-specific weights as the corrections are summed. Here they propose a method of computing weights such that when applied to the corrections within the bilinear form, the bilinear form maps onto the same value as the linear form on the right hand side of the finite element problem maps to. Computing these weights requires an additional matrix-vector product applied to the entire grid and additional dot product per grid level. The resultant system can be singular however, so this is not a perfect solution. Recent work by Vangara, Kashi and Nadarajah [77] introduces a level-specific scaling parameter to recover multiplicative

multigrid convergence rates. The computed damping parameter is the result of two additional inner products per grid level and minimises the difference between the residual from multiplicative multigrid smoothing and the actually performed additive smoothing process.

A widespread additive multilevel approach, that modifies coarse equations with a strong focus on parallelism, is the original "BPX" algorithm. This is an early example of an additive multigrid scheme, and remains a widely used choice of preconditioner [67], [78]. The main features that separate BPX from other additive schemes are the specific choice of coarse grid spaces and coarse grid equations. BPX is built on parallel (nested) subspace corrections. Coarse grid corrections are merely damped restricted residuals, rather than the result of a coarse solve. Restricted residuals are weighted by an approximated diagonal matrix. Single iterations/corrections can therefore be computed more quickly; however, more iterations may be required for convergence—ideally these factors balance to reduce time-to-solution. The coarse grid equations are defined simply—coarse grid corrections only use diagonal scaling of the fine grid input. This has drawn comparison between BPX and Multilevel Schwarz Algorithms. BPX has been shown to be equivalent to the Multilevel Diagonal Scaling (MDS) (which is a modification of Zhang's Multilevel Additive Schwarz (MAS) method [79]). The stability of BPX has been improved by updating vertices according to a colouring scheme [80]. BPX also further limits the domain of the coarse correction spaces to improve stability. Fine grid smoothing is only performed on regions that do not hold coarse grid points. The colouring scheme is said to be optimal for certain classes of equations.

Three coarse grid equation definitions that are of interest to our work are: the "BPX" algorithm [67], [78], specifically the MDS variant; Dendy's BoxMG [21], [70], [71]; and algebraic multigrid. BoxMG changes the definition of coarse grid equations through changing the intergrid transfer operators. How stability changes between these three methods for additive multigrid schemes is explored by Grauschopf et al [22]. They compare the three alternatives: Here, BoxMG is shown to be generally

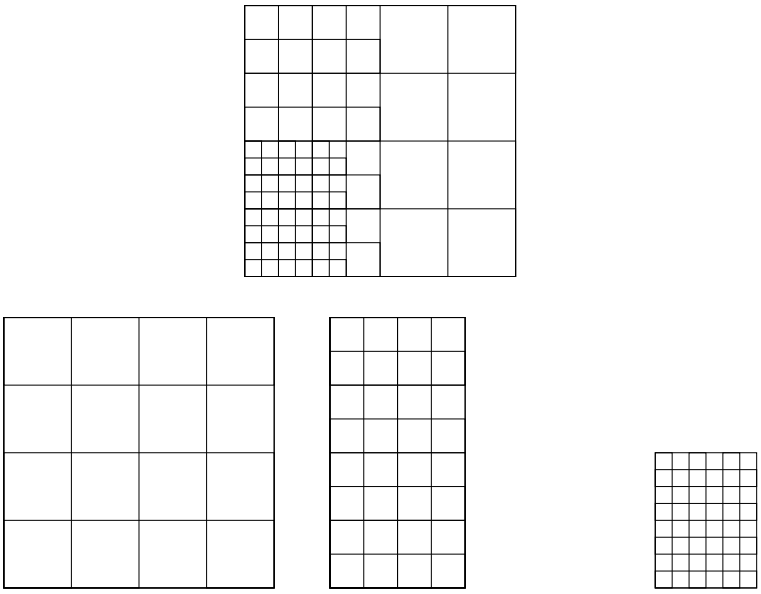

Figure 3.5: A composite grid composed of multiple different mesh resolutions (top). The three different regular subgrids who's conjunction forms the composite grid (bottom).

competitive with AMG as both sets of coarse grid equations have lower condition numbers than MDS and thus converge in fewer iterations; the simpler construction of MDS is shown to worsen the convergence rate. BoxMG and AMG are also shown to be effective for more challenging setups.

3.2.5 The FAC family of solvers

The Fast Adaptive Composite grids (FAC) family of solvers [38] are collection of methods that serve as a building block for a lot of our work. We build upon the damping term from AFACx [81], [82] in particular. What follows is a brief explanation of a few FAC solvers and their underpinning ideas.

Within a multigrid context, a simple mesh storage solution can become a concern with the introduction of adaptivity as multigrid requires a hierarchy of meshes and associated coarse grid equations. In a structured grid, the fine grid discretisations embed within the coarse grid discretisations. Fine grid operators therefore, also embed within coarse grid operators. The different grid levels are coupled, so a method of handling this indirect coupling is required. The FAC family of solvers

write fine grid problems as subproblems that are solved separately. All grids store full solution values and compute corrections for their respective grids. Fine grids are subproblems and grid corrections are solved sequentially. MultiLevel Adaptive Technique (MLAT) [20], [83], [84] and Fast Adaptive Composite grids (FAC) [17], [38], [85] are multilevel adaptive schemes—they use multiple grid levels with different resolutions that are handled sequentially/separately. An example of the overall composite grid and how that example grid is composed of multiple subgrids is shown in Fig. 3.5. This yields a hierarchical generating system rather than a basis.

FAC describes a multiplicative multigrid scheme over a hierarchical system. Starting from the finest grid: the residual equation is determined there and then smoothed; the residual is then re-computed and restricted to the next coarser level. It continues recursively. This is a multiplicative multilevel (multigrid) scheme. Early FAC papers operate on the assumption of a small set of reasonably fine grids and leave the implementation open of what solver to use on the individual grids. Some explicitly speak of FAC-MG if a multigrid cycle is used as the iterative smoother per level. Here we refrain from such details and consider fast adaptive composite grid as a multiplicative scheme overall which can be equipped with simple single-level smoothers. The first FAC papers [38] acknowledge difficulties for operators along the resolution transitions. FAC traditionally uses a top-down traversal [15]: The cycle starts with the coarsest grid, and then uses the updated solution to impose Dirichlet boundary conditions on hanging nodes on the next finer level. This inversion of the grid level order continues to yield a multiplicative scheme as updates on coarser levels immediately propagate down and as all steps are phrased as residual update equations.

FAC relies on spatial discretisations that are conceptually close to space-trees. Both approaches thus benefit from structural simplicity: As the grid segments per-level are regular, solvers (smoothers) for regular Cartesian grids can be (re-)used. As the grid resolutions are aligned with each other, hanging nodes can be assigned interpolated values from the next coarsest grid with a geometrically inspired prolongation. As all

grid entities are cubes, squares or lines, all operators exhibit tensor-product structure. FAC's hierarchical basis differs from textbook multigrid [13] for adaptive meshes: The fine grid smoothers do not address the real fine grid, but only separate segments that have the same resolution. The transition from fine to coarse grid does not imply that the number of degrees of freedom decreases. Rather, the number of degrees of freedom can increase if the fine grid holds a localised AMR region with many refinements. It is obvious that this poses challenges for parallelisation. Different grid levels produce corrections that act upon the same regions of space and therefore the same fine grid degrees of freedom—this creates a coupling between the grid levels. Different grid levels cannot be handle totally in parallel in a naive manner, as information propagates between them. If updates do not flow, then different grid levels will produce redundant updates for the same grid points—possibly overcorrecting.

FAC can be mechanically augmented into a version with additive corrections. The hierarchical generating system renders this endeavour straightforward. Plain additive multigrid on a FAC data structure once again yields a non-robust solver that tends to overcorrect [15], [32]. The hierarchical basis approach starts from the observation that the instabilities within the generating system are induced by spatially coinciding vertices. Therefore, it drops all vertices (and their shape functions) on one level that coincide with coarser vertices. The asynchronous fast adaptive composite grid (AFAC) solver family finally modifies the operators to anticipate overshooting [16], [81]. BPX may be read as particular modification of additive multigrid and AFAC as a generalisation of BPX, as stated by Jimack and Walkley [86]. AFAC was historically served in two variants [15]:

AFACc simultaneously determines the right-hand side for all grid levels ℓ. Before it restricts the fine grid residual to a particular level ℓ, any residuals on vertices spatially coinciding with vertices on the level ℓ are instead set to zero. They are masked out on the fine grid. This effectively damps the correction equation's right-hand side. If we applied this residual masking recursively—a discussion explicitly not found in the original AFACc paper where only the points are masked which coincide with the

target grid—i.e. if we constructed the masking restriction recursively over the levels instead of in one rush, then AFACc would become a hybrid solver between additive multigrid and the hierarchical basis approach.

AFACf goes down a different route: The individual levels are treated independently from each other, but each level's right-hand side is damped by an additional coarse grid contribution. This coarse grid contribution is an approximate solve of the correction term for the particular grid. AFACf solves all meshes in parallel and sums up their contributions, but each mesh has its contribution reduced by the local additional coarse grid cycle. The resulting scheme is similar to the combination technique as introduced for sparse grids [87]: We determine all solution updates additively, but remove the intersection of their coarser meshes.

These two modifications allow great parallelism for FAC, but the full solves on all coarse and auxiliary grid levels in AFAC are still expensive. AFACx is an additional modification that swaps the full solves for simple smoothing steps instead, this is observed to reduce the workload significantly but not noticeably harm the rate of convergence. This is similar to standard additive multigrid and BPX, where corrections on each grid level arise just from smoothing; however, the auxiliary grid correction provides a damping term that isolates corrections to a specific grid level to prevent overshooting. We introduce a smoother, $\tilde{M}_{\ell-1}^{-1}$, for this auxiliary grid level. A $\tilde{}$ denotes that this belongs on the auxiliary grid. This exists on a grid with increased mesh spacing, hence we have changed the index—even though it is formally a correction that belongs to level ℓ.

$\tilde{M}_{\ell-1}^{-1}$	Smoother on an auxiliary equation level $\ell-1$

For c_ℓ, an arbitrary correction on level ℓ, auxiliary smoother $\tilde{M}_{\ell-1}^{-1}$ and arbitrary intergrid transfers R and P, AFACx produces the per-level correction

$$\begin{aligned} c_\ell &= \underbrace{P\tilde{M}_{\ell-1}^{-1}Rr_\ell}_{\text{Initial guess}} + \underbrace{M_\ell^{-1}Rr_\ell - M_\ell^{-1}A_\ell P\tilde{M}_{\ell-1}^{-1}Rr_\ell}_{\text{Coarse grid smoothing}} - \underbrace{P\tilde{M}_{\ell-1}^{-1}Rr_\ell}_{\text{Damping term}} \\ &= M_\ell^{-1}Rr_\ell - M_\ell^{-1}A_\ell P\tilde{M}_{\ell-1}^{-1}Rr_\ell. \end{aligned}$$

Note that the initial guess and damping term effectively cancel. AFACx again operates on a composite grid and appropriate hierarchies of both correction grids and auxiliary correction grids. To improve robustness relative to additive multigrid, an additional sequential component is introduced between the grid levels—the auxiliary smoothing step is performed before the smoothing process on the original coarse grid levels. A single residual vector is computed, on the "composite" grid, and this residual is restricted to all grid levels, including the auxiliary grid level. The auxiliary grid is smoothed using zero as initial guess. This computes a correction that is projected to the original coarse grid hierarchy. Each auxiliary grid is built over an existing coarse grid space and projects its correction to that single grid space. This projected correction serves as initial guess for smoothing on the coarse grid level. After smoothing the auxiliary correction is used as an additive damping term to prevent overcorrections. This means that corrections components on a coarse grid level are isolated from the components they would share with the next coarsest level.

3.2.6 Additional parallel multigrid implementations

Beyond additive multigrid, there are more general techniques to increase the potential for parallelism within multigrid or scale multigrid to very large machines. None of these ideas modify the coarse equations, intergrid transfers or coarse grid spaces specifically—the following schemes either do a combination of those ideas or something completely novel. We briefly outline a few of them here and how they tie into our work.

Parallel Superconvergent MultiGrid (PSMG) [88] introduces multiple grid spaces. It is a multiplicative scheme, and, similar to AFACx, introduces additional work in the form of supplementary grids. Multiple coarse grids for the same fine grid are used, and this produces multiple corrections for the same resolution with the additional coarse grids reducing the number of idle cores relative to regular multiplicative multigrid when computing the coarsest corrections. Theoretically improved convergence may

be seen from these additional corrections, but this was not experimentally verified. Rather than increasing the workload to take advantage of all parts of the system communication can instead be minimised. Mult-additive [89] follows this idea and removes the sequential processing of coarse grids entirely. Multiplicative coarse grid corrections are written in purely additive semantics through the use of smoothed interpolation operators, so that the convergence rate of multiplicative multigrid is maintained, but the communication cost is closer to additive multigrid. Mult-additive modifies the intergrid transfers using a similar philosophy to smoothed aggregation multigrid and modifies itself to more closely resemble multiplicative multigrid.

Changing the smoother itself is a concept used in "Asynchronous Multigrid" [90]. Wolfson-Pou and Chow have implemented mult-additive and AFACx (and conventional multiplicative multigrid) using asynchronous smoothers, An asynchronous smoother is one where the number of smoothing steps on differing grid levels and different regions of the same grid level are not guaranteed to be synchronised, i.e. the same. Mult-additive was shown to be the most effective implementation they tested under various metrics—being both the quickest in terms of wallclock time and generally being minimal in the number of iterations required for convergence. AFACx in general required additional iterations to converge and a proportionally increased run-time.

There have been many multigrid schemes designed for large scale runs—both for large core counts and for large numbers of degrees of freedom. Gmeiner et al. [91] have seen success in scaling multiplicative multigrid to large setups (300,000 cores and problem sizes of 10^{12} unknowns), but their implementation is only for simple, purely geometric setups. More complicated conditions were tackled by Lin et al. [92] using algebraic multigrid. Effective scaling is seen up to 1.6 million cores for the Poisson equation. However, common to many algebraic multigrid implementations, the setup is a non-negligible factor. Impressive scaling can instead by achieved with a conjunction of different solvers rather than solely one, for example, May et al. [93] use a geometric multigrid solver on the finer levels, an algebraic solver on the

next coarsest level and Krylov methods on the coarsest grid; this implementation scales well for the tested setups, but only up to a total of 4000 total cores. The multigrid component is also minimised and does not use a large number of grid levels. Work by Rudi [94] also merges many solver types into one. Multiple forms of coarsening—h-coarsening, p-coarsening and then a switch to an algebraic multigrid solver—are used. Their solver shows promising scalability (up to 1.5 million cores and for ill–condition problems); however, it is only used as a preconditioner rather than as a solver in and of itself.

3.2.7 HTMG and FAS on space-trees

We use adaptive grids throughout our work—to simplify the implementation of them we borrow ideas from full approximation storage (FAS) and use the hierarchical transformation multigrid (HTMG) implementation. Although the implementation of multigrid on adaptive meshes is, in principle, straightforward, implementational complexity arises along resolution transitions. Weights associated to the vertices change semantics once the comparisons for vertices on a level ℓ are made between points that belong to the fine grid and those that belong to refined vertices: The latter carry a nodal solution representation, i.e. a scaling of the Finite Element shape functions, while the former carry correction weights. In classic multigrid starting from a fine grid and then traversing correction levels, it is not straightforward how to handle the vertices on the border between a fine grid region and a refined region within one level. They carefully have to be separated [32], [33].

One solution to address this ambiguity relies on FAS [13]. Every vertex holds a nodal solution representation. If two vertices v_ℓ and $v_{\ell+1}$ from coarse level ℓ and fine level $\ell+1$ spatially coincide, the coarser vertex holds a copy of the finer vertex: In areas where two grids overlap, the coarse grid additionally holds the injection $u_\ell = I u_{\ell+1}$ of the fine grid. This definition exploits the regular construction pattern of space-trees. Vertices in refined areas now carry a correction equation, plus the

injected solution rather than a sole correction. The injection couples the fine grid problem with its coarsened representation and makes this representation consistent with the fine grid problem on adjacent meshes which have not been refined further. Griebel's HTMG [95] is an effective implementation of the full approximation storage scheme. It also relies on the assumption/approximation that all operators can be approximated by Ritz-Galerkin multigrid $RA_{\ell+1}P = A_\ell$. Injection $u_\ell = Iu_{\ell+1}$ allows a rewrite of each and every nodal representation into its hierarchical representation $\hat{u}_\ell = (id - PI)u_\ell$. A hierarchical residual $\hat{r}$ is defined in the expected way. This yields the modified multigrid equation:

$$\begin{aligned} A_\ell\,(u_\ell + c_\ell) &= A_\ell u_\ell + A_\ell c_\ell = R\hat{r}_{\ell+1} \\ &= R\,(b_{\ell+1} - A_{\ell+1}(u_{\ell+1} - PI_{\ell+1})) \\ &= R\,(b_{\ell+1} - A_{\ell+1}\hat{u}_{\ell+1})\,, \end{aligned} \tag{3.2.2}$$

i.e. per-level equations

$$A_\ell u_\ell = \begin{cases} b_\ell & \text{on the fine grid (regions)} \\ b_\ell = R\hat{r}_{\ell+1} & \text{on the coarse grid (regions) with } \hat{r}_{\ell+1} = b_{\ell+1} - A_{\ell+1}\hat{u}_{\ell+1}. \end{cases}$$

To the smoother, u_ℓ resulting from the injection serves as the initial guess. Subsequently it determines a correction c_ℓ. This correction feeds into the multigrid prolongation.

3.3 Hardware and implementation specifics

3.3.1 Memory accesses

We seek to modify the assembly process and write a new implementation and also develop a new family of solvers for our solver suite. Therefore we must take into account the underlying hardware for our implementation. We briefly cover the key features that we incorporate into our algorithmic design.

Quite intuitively, one possible limit on how quickly a solver can complete a solve is the hardware used—faster hardware therefore has historically guaranteed faster times to solution. This observation no longer strictly holds. Processor clock speeds are increasing at a rate greater than the rate of increase in memory bandwidth [18]. Processors complete local operations and are then required to wait for subsequent data that is still held in memory, therefore faster clock speeds do not always lead to reduced time-to-solution for a fixed solver.

Minimising memory accesses in iterative algorithms is therefore required to optimise for modern hardware. The underlying multigrid algorithm is optimal in terms of compute steps, the associated memory access patterns are often not. Associated matrices can be sparse, leading to scattered memory accesses and poor data access patterns. Many mesh structures use indirect memory accesses which increases memory latency. High memory latency and low memory bandwidth means low arithmetic intensity in multigrid implementations. Pipelining, wherein operations are blocked such that data can be streamed into the core/cache from main memory and not delay communications, is well trodden ground. Formulations of the pipelined conjugate gradient method are a good example of this [96], [97]. Similar principles have been applied to multilevel methods—in particular multigrid [98]–[100].

3.3.2 Element-wise operator decomposition and storage

Within our target implementation, all of our operators are decomposed and applied in an element-wise fashion to reduce memory access costs—this follows on from existing multigrid implementations [100], [101] and standard finite element cell-wise operator decompositions [102]. Assembling a matrix as an explicit data structure both incurs a large memory cost when storing elements and an increase in latency when fetching elements. The latter is often due to poor memory access patterns. Stencils restructure the matrix, matrix entries are stored in memory locations that are local to the degrees of freedom they effect. A vertex in a mesh therefore need

only store matrix entries that impact it directly. The matrix is decomposed into rows that are then stored in individual vertices, this is nothing new but serves as motivation for further decompositions.

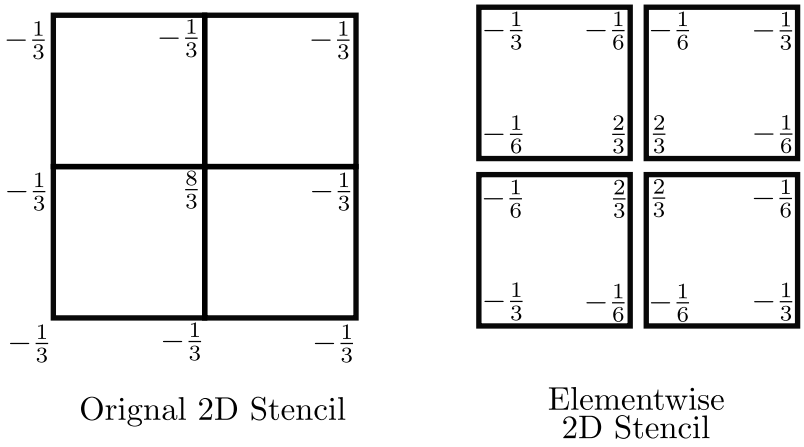

Figure 3.6: Decomposition of a nodal stencil into element-wise stencils on cells.

A stencil can be further decomposed into components that are separately processed by neighbouring cells. An example decomposition can be seen in Fig. 3.6. Stencils are stored nodally and the element-wise decomposition is computed on demand. A decomposition such as this is not unique. A stencil is additively split into components centred around the cells (or elements in a finite element sense) that are adjacent to the vertex on which the original stencil was centred. The element-wise decomposition stores weights for all vertices adjacent to the chosen cell/element. The decomposition can be applied to all cells independently and the results summed—as the element-wise stencils result from an additive decomposition this produces the same result as applying the nodal stencil.

For a matrix-free implementation, such stencils are recomputed on-the-fly, but as soon as permanent recomputation of stencils becomes too expensive, the assembly matrices must be stored. To avoid the maintenance of an explicit matrix data structure, some schemes [32], [33] store the local element matrices directly within the tree cells, i.e. embed the stencils into the cell stream. This avoids the memory overhead of matrix data structures requiring meta data, sparsity pattern information

and so forth, but it still has to pay for all actual matrix entries. They just are encoded within the mesh rather than within a dedicated data container. Similar arguments can be made for algebraic intergrid transfer operators—purely geometric operators can be computed on-demand and require no precomputation. Each vertex stores its element-wise operator parts from A. Each coarse grid vertex carries its prolongation and restriction operator plus its stencil.

3.3.3 Single-touch

Following on from the ideas of Reps and Weinzierl [32], all of our algorithms are written to be single-touch. A single-touch algorithm is an iterative algorithm that aims to read to and writes from each memory location only once per iteration. This can be verified by measuring cache hits and cache misses. If data is fetched only once and stays in cache, then an algorithm can be assumed to be single-touch. There are two main factors to account for when writing an iterative algorithm to be single-touch: Firstly, the mesh/elements must be traversed in a manner that allows for a single-touch handling of degrees of freedom. Secondly, degrees of freedom must themselves be handled in a single-touch manner. We briefly outline an ordering of elements for a mesh traversal, using a depth first traversal, that allows for a single-touch implementation. Then we cover the specific reads/writes on degrees of freedom that allow an additive multigrid iteration to be single-touch.

Traversing the mesh in a single-touch manner In Reps and Weinzierl's work, they developed an effective multigrid pipeline, which starts from an element-wise traversal of the set of meshes. $\{\Omega_0, \Omega_1, \ldots, \Omega_{\ell_{\max}}\}$ defines an ordering on the levels of the mesh, and also provides a partial ordering of the cells within the mesh—it yields a stream of cells. The partial ordering does not define an order on cells on the same level, it only strictly defines an ordering between levels. There is no order constraint on the cell enumeration within the stream for a specific level. We reiterate that in both their work and our work, we construct the matrix using a nodal assembly but

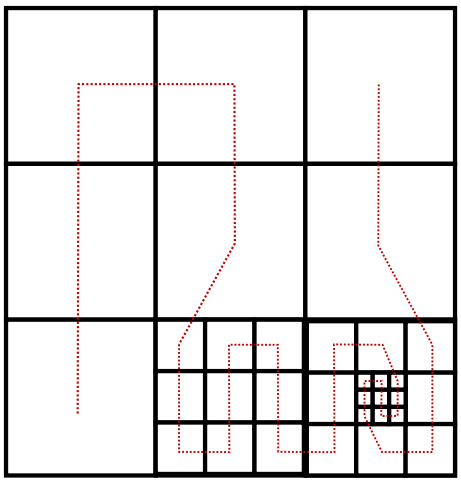

Figure 3.7: Traversal order of a sample mesh that has been adaptively refined. The coarse mesh is traversed via a space-filling Peano curve. Refined regions are expanded via a depth-first search when encountered, the refined patches are recursively traversed via Peano curves.

mat-vec products are handled via an element-wise assembly/application. When we handle the matrix using such an element-wise decomposition, as in Section 3.3.2, we only treat each cell, and the associated vertices, once per grid traversal. We therefore seek a method handling these cells such that they're only loaded into main memory once and similarly adjacent degrees of freedom are only written to once. For a realisation of such an element-wise handling, possibly in a (dynamically) adaptive scheme, there are many examples where the mesh is treated via a depth-first (DFS) traversal of a space-tree [32], [33], [44], [100]. Each level's cells, can therefore, be organised along a space-filling curve [42]. An example traversal can be seen in (Fig. 3.7). For modern hardware, data accesses are often a limiting factor. Reading to and writing from main memory slows solves tremendously. We therefore introduce the idea of single-touch—we limit both of these operations to once per grid traversal per-vertex.

Definition 3.3.1. Single-touch: Variables in main memory are read to and written from only once per grid traversal. They are loaded into cache once and only written back once.

With a naive element-wise decomposition, where each vertex is handled separately, we would read/write from each memory location on a Cartesian mesh 2^d times each time we perform an update. Each vertex is adjacent to 2^d cells in a Cartesian mesh and partial updates therefore require 2^d cells accesses and with each access causing 2^d data writes to all relevant degrees of freedom. An element-wise matrix for a cell can be constructed by decomposing the stencils of all adjacent vertices. The partial stencils can then be processed for a cell in a single matrix-vector product, rather than requiring multiple evaluations. Thus allowing degrees of freedom to remain in cache during the traversal.

The combination of DFS with space-filling curves, means the tree traversal is weakly single-touch with respect to the vertices: Vertices are loaded when an adjacent cell from the space-tree is first entered. They are "touched" for the last time once all 2^d adjacent cells within the space-tree have been left due to recursion backtracking. In-between, they reside either on the call stack or can be temporarily stored in stacks [42]. The call stack is bounded by the depth of the space-tree—it is small—while all temporary stacks are bounded by the time in-between the traversal of two face-connected cells. The latter is short due to the Hölder continuity of the underlying space-filling curve. Hanging vertices per grid level, i.e. vertices surrounded by less than 2^d cells, are created on-demand on-the-fly. They are not held persistently. It is therefore assumed that all data remains in the caches [44], [100].

Accessing degrees of freedom in a single-touch manner An additive algorithm can be written to be single-touch. An additive correction scheme first computes the residual on the fine grid, restricts it to the right-hand side of all coarser correction levels, computes the update on a coarse grid level, and finally projects this coarse update to the fine grid. This implies a fine-to-coarse and subsequent coarse-to-fine flow of information, which does not fit neatly into a DFS traversal of a grid—which first unfurls the mesh coarse-to-fine and then backtracks fine-to-coarse. We instead offset this by half an iteration—therefore, the data flow now follows the

mesh traversal. Due to the mesh traversal, we enforce a partial ordering—a fine grid residual for a vertex is restricted to the next coarser grid level after all other computations for the fine grid vertex have been performed. Therefore, locally the fine grid must be handled before the coarse.

Algorithm 4 Outline of single-touch additive multigrid. sc is the summed coarse grid correction contributions. sf is the summed fine grid correction contributions. $S(u_\ell, b_\ell)$ is the smoother applied to u_ℓ We invoke the cycle passing in the coarsest grid ℓ_{min}.

```
function additiveMG(ℓ)
    sc_ℓ ← sc_ℓ + P^ℓ_{ℓ−1} sc_{ℓ−1}        ▷ Prolong contributions from coarse grid
    u_ℓ ← u_ℓ + sc_ℓ + sf_ℓ                  ▷ Anticipate fine grid smoothing
    û_ℓ ← u_ℓ − P^ℓ_{ℓ−1} u_{ℓ−1}            ▷ Determine hierarchical solution
    if ℓ < ℓ_max then
        additiveMG(l + 1)
    end if
    r_ℓ ← b_ℓ − A_ℓ u_ℓ                      ▷ Compute residual
    r̂_ℓ ← b_ℓ − A_ℓ û_ℓ                      ▷ Compute hierarchical residual
    sc_ℓ ← ωS(u_ℓ, b_ℓ)                      ▷ Perform coarse smoothing
    if ℓ > ℓ_min then
        b_{ℓ−1} ← R^{ℓ−1}_ℓ r̂_ℓ
        sf_{ℓ−1} ← I(sf_ℓ + sc_ℓ)
    end if
end function
```

Similarly, a full approximation storage (e.g. HTMG) sweep can not straightforwardly be realised within a single DFS grid sweep [32]: The residual computation propagates information bottom-up, the corrections propagate information top-down, and the final injection propagates information bottom-up again. This yields a cycle of causal dependencies. Additive cycle's smoothing steps are thus offset by half a grid sweep again: Each grid sweep, i.e. DFS traversal, evaluates two mat-vecs—of FAS, of the hierarchical transformation multigrid—but does not perform the actual updates. Instead, correction quantities–for fine grid correction, coarse grid corrections and local corrections—are bookmarked as additional attributes within the vertices while the grid traversal backtracks, i.e. returns from the fine grids to the coarser ones. Their impact is added to the solution throughout the downstepping of the subsequent tree sweep. Here, the prolongation can also be evaluated. Restriction of the residual to the auxiliary right-hand side and hierarchical residual continue to be the last action

on the vertices at the end of the sweep when variables are last written/accessed. These plug into a recursive function's backtracking, all right-hand sides are thus accumulated from finer grid levels by the final time a vertex is touched through a multiscale grid traversal. All updates are computed but not applied. Only one tree traversal is required per V-Cycle (plus one kick-off traversal). The helper variables pick up on ideas behind pipelining and are a direct translation of optimisation techniques proposed by Reps and Weinzierl. Per traversal, each unknown is read into memory/caches only once. It is a "single-touch" implementation. The exact steps for this are explicitly shown in Algorithm 4 (this is reprint of the algorithm from [32] shown for clarity).

Chapter 4

Lazy Stencil Integration

In the preceding chapter, we laid out the groundwork and the existing work that our ideas build upon. Now we move onto the first of our novel ideas. We introduce our method of asynchronous equation construction—this is a method of integrating fine grid stencils using a lazy evaluation. The computation of the exact—"true"—stencil is delayed until later in the solve than with conventional assembly and we use initial approximations of the stencil in early solver iterations. In this chapter we explain how we can delay the equation construction without also delaying the multigrid solve. We begin this chapter by explaining how to construct multigrid equation assembly as a series of sequential operations—this is then further broken down so the computation of an individual stencil is the result of an iterative series of parallelisable tasks. Latter parts of the chapter give high concept pitches of how our ideas can integrate with different aspects of a multigrid cycle. The general theory is covered in this chapter but not our implementation. A detailed explanation of our target implementation is instead given in Chapter 6.

The following chapter is modified from text that was previously published in [2], [3]. The introductory covering of terminology (Section 4.1) and remarks on dynamic adaptivity (Section 4.8) are modified from the earlier paper—"Lazy Stencil Integration in Multigrid Algorithms". The intermediate sections (Section 4.3-4.7)—where we iteratively construct what we mean by an adaptive stencil integration—and the terminology section are edited from the latter paper—"Delayed approximate matrix assembly in multigrid with dynamic precisions".

4.1 Outline

Solving the matrix equation $Ax = b$ is costly for many reasons. Tackling a problem across multiple scales is a seminal idea to reduce the cost—solvers that implement this principle can thus yield optimal complexity for some problems. Optimality here, however, refers exclusively to the solve process. The setup for such solves is still not provided for free, as the assembly process for all the required data structures is non-trivial. We have to numerically integrate all stencils on the fine grid in order to construct the matrix equation. For complex equations, such as those with non-constant material parameter ϵ, this numerical integration is a complex sequence of operations and therefore costly. Some multigrid implementations require the coarse grid spaces themselves to be computed in the assembly process. All multigrid implementations, however, require a coarse grid assembly phase, i.e. construction of coarse grid stencils and intergrid transfer operators for all coarse grid levels in the hierarchy. Even though we work in a best case scenario—we use space-trees which define a set of coarse grids known a priori—we have to accept that the operator construction (assembly) in robust multigrid makes up for a non-negligible part of the total run-time [33]. Assembly is painful. It is the character of the assembly that complicates the situation further: Often, we are constrained by memory characteristics, specifically bandwidth and memory latency. Data from memory cannot be streamed to cores at a rate that matches current processing speeds. Cores are therefore idle. As this gap between compute power and memory bandwidth widens [34], the assembly becomes harder and harder to scale.

A cheap method of constructing operators is to extract geometric information from the underlying PDE. This use of geometric information leads to operator rediscretisation on the coarse grids and d-linear intergrid transfer operators, where d is the dimension. Both choices reduce the assembly overhead. Rediscretisation is the process of assembling coarse grid stencils using a discretisation process on the coarse grid space. The coarse grid assembly process is therefore similar to assembly on the

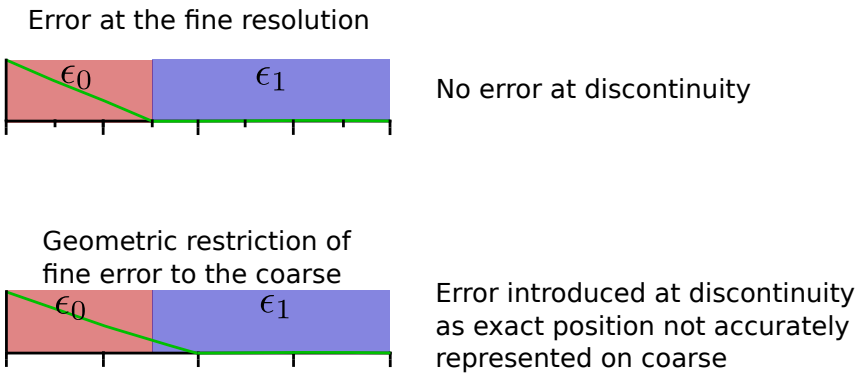

Figure 4.1: Top: Sample error on a fine grid. Bottom: The same error is restricted to the coarse grid using geometric operators. If the coarse grid exactly removed the coarse representation of the error, it would introduce error on the fine grid.

fine grid. Yet, the performance and robustness of rediscretised equations deteriorates when we face non-constant ϵ [35], [40]. See Fig. 4.1, where error is geometrically restricted to the coarse grid. If this error is exactly removed, it would introduce error on the fine grid and cause oscillatory behaviour in fine grid solution values. Therefore, an algebraic solver that uses the fine grid stencils to construct intergrid transfer operators and the resultant coarse grid equations may become a requirement. Typically, assembly refers to exclusively constructing the fine grid equations, however, as we construct Ritz-Galerkin operators for coarse grid equations, we expand our use of the term to also include the construction of coarse grid operators and intergrid transfer operators.

When constructing coarse grid equations, effective methods should take into account the computational cost of assembly and the required storage. Geometric solvers can require a minimal amount of storage because coarse grid operators are able to be hard-coded. They require no additional fine grid information. This incurs additional computational costs, however, as they must be recomputed each time they are required. Recomputing coarse grid equations on-the-fly can become infeasible when there are dramatic jumps in the fine grid material parameter. A coarse mesh will therefore encapsulate more of these jumps, which further complicates the coarse grid

assembly. Computing and then explicitly storing coarse grid equations and intergrid transfer operators, as is usually the case for algebraic multigrid, can drastically increase memory requirements. A hybridisation of algebraic and geometric multigrid is a method of balancing these concerns—performing algebraic multigrid on the coarser grid levels and geometric on the finer is one such example [73], [93]. These concerns motivate our methodology.

4.2 Problem characteristics

We state the characteristics of multigrid as a series of operations, so we can explain how we reduce the cost of implementing it. Classic multigrid, composed of a separate construction and solve phase, can be read as a sequence of activities

$$(\mathcal{S} \circ \ldots \circ \mathcal{S}) \circ \ldots \circ \mathcal{A}_{\ell-3} \circ \mathcal{A}_{\ell-2} \circ \mathcal{A}_{\ell-1} \circ \mathcal{A}_{\ell}^{(\text{geo})}(\epsilon). \tag{4.2.1}$$

$\mathcal{S}$	Multigrid smoother iteration across all levels
$\mathcal{A}_\ell$	Generic assembly process for level ℓ
$\mathcal{A}_\ell^{(\text{geo})}$	Geometric assembly process for level ℓ

We start with the integration of a discretisation of (1.1) on a fine grid of mesh size h identified by its level ℓ. $\mathcal{A}_\ell^{(\text{geo})}$ yields the resulting discretisation matrix A_ℓ. This is the assembly process in a finite element sense. We reiterate that the process determines a stencil (3.1.2), i.e. integrates

$$\int_{\Omega_h} \epsilon \left(\nabla u, \nabla \phi\right) \; dx = \sum_{c \in \Omega_h} \int_c \epsilon \left(\nabla u, \nabla \phi\right) \; dx,$$

for all cells in the mesh. It is a process based on the geometry. It incorporates geometric information directly. The material ϵ directly enters the arising linear equation system A_ℓ.

Definition 4.2.1. Task (fine grid): for a fine grid cell a task is defined as an individual integration over an element in a finite element sense.

With a fine grid matrix to hand, or rather a method constructing this matrix, we now detail the construction of the remaining multigrid operators. The coarse grid equations could likewise be defined geometrically and also constructed via (numerical) integration. Quite simply, $\mathcal{A}_{\ell-1}$ would be $\mathcal{A}_{\ell-1}^{(\text{geo})}$. The restriction and prolongation operators, R and P, are also defined geometrically and can be computed on-the-fly. Alternatively we can use an algebraic construction. This is a two-step process for each level. Here, $\mathcal{A}$ is defined in a multigrid sense compared to $\mathcal{A}^{(\text{geo})}$ on the finest grid, i.e. we assume that $\mathcal{A}$ is defined using Ritz-Galerkin: $A_{\ell-1} = RA_\ell P$. A Ritz-Galerkin assembly process requires the transfer operators to be constructed prior to the construction of the coarse grid equation. For simplicity of our notation here, we assume that all coarse grid meshes are known; the identification of coarse grid meshes becomes a separate assembly step otherwise. This summarises the computations that make up the assembly process for our multigrid algorithm. Each grid level (sans the finest) requires three matrices—the coarse grid equation A_ℓ and the two intergrid transfer operators R and P—to be constructed, and possibly stored.

Definition 4.2.2. Task (coarse grid): For a coarse grid cell a task is defined as any individual process that computes the three element-wise coarse grid operators.

$\mathcal{S}_\ell$	Multigrid smoother iteration on single grid level

Once all operators are set up, each multigrid cycle $\mathcal{S}$ decomposes into a series of actions $\mathcal{S}_\ell$ on specific grids. These actions are the smoothing steps. They can be further broken down into sub-actions that operate on specific elements or local regions of a grid. Depending on how we arrange and design these sub-actions, the overall scheme denotes an additive or multiplicative cycle.

A multigrid cycle is a series of actions that operate on grids in a designated order. Different multigrid cycles run through the grid hierarchy, thus enacting these actions, differently. In multiplicative this is serial; in additive this is parallel. For specific grid levels in a two grid cycle, a multiplicative $V(1,1)$-cycle in this notation becomes $\mathcal{S}_\ell \cdot \mathcal{S}_{\ell-1} \cdot \mathcal{S}_\ell$ (the smoothing steps for all levels are handled sequentially), while an additive cycle is $\mathcal{S}_\ell + \mathcal{S}_{\ell-1}$ (the smoothing steps for all levels are handled independently). While the number of resolution levels determines the length of the $\mathcal{A}$ sequence as well as level updates, the solve itself is iterative: We therefore use the generic symbol $\mathcal{S}$, which summarises a whole multigrid solve (cycle), multiple times. The total number of $\mathcal{S}$ applications is typically steered by the residual. We terminate when the residual normalised by its initial value falls below a threshold, i.e. when the error has been diminished by this factor.

Our work proposes a lazy element-wise assembly based on tasks: It starts from a trivial integration of the weak formulation of (1.1) where we sample ϵ in the centre of the cell once, to get the constant ϵ_c per cell. That is we initially approximate the weak formulation as

$$\sum_{c \in \Omega_h} \int_c \epsilon \left(\nabla u, \nabla \phi\right) \; dx = \sum_{c \in \Omega_h} \epsilon_c \int_c \left(\nabla u, \nabla \phi\right) \; dx,$$

For n cells, the assembly generates n cell-wise tasks that each contribute to the global assembly matrix. Each task is a low accuracy integration over that specific cell. In parallel to the actual solve, we then improve the quadrature per stencil and compute new stencil entries with better and better accuracy. With i being a subcell within cell c, we now approximate the weak formulation as

$$\sum_{c \in \Omega_h} \int_c \epsilon \left(\nabla u, \nabla \phi\right) \; dx = \sum_{c \in \Omega_h} \sum_{i \in c} \epsilon_i \int_i \left(\nabla u, \nabla \phi\right) \; dx.$$

Each cell therefore spawns an iterative series of tasks. Whenever we need a stencil and its next higher accuracy is not yet available, we continue with the old "low-accuracy" one. It is a greedy process not delaying the solve. The assembly and smoothing steps are interleaved and performed concurrently. As soon as two subsequent numerical

integrations do not yield a large difference anymore, we stop the series of tasks for this particular element. It is an adaptive process. As new stencils become available, we restrict their influence to the next coarser level in the next multigrid cycle. It is an iterative process where the algebraic coarse-grid computation and Ritz-Galerkin construction incrementally push the operator information up the resolutions. The operators ripple through the hierarchies.

4.3 Numerical computation of stencils in a task language

We have given a high level overview of the activities within a multigrid solver, and now we further break these down into our defined tasks that we reorder. We return to our interpretation of multigrid operations as a series of actions upon grid levels: $\mathcal{S}^{(DoF)}$ specifically denotes an individual smoothing update of the multigrid algorithm i.e. for one particular vertex/degree of freedom only; $\mathcal{A}^{(\text{geo})}$ is again a (geometric) assembly task for a single element that is adjacent to the designated vertex/degree of freedom. We omit the level indexing and work exclusively on the fine grid level here.

A geometric, matrix-free implementation of multigrid then issues

$$\mathcal{S}^{(DoF)} \circ \underbrace{(\mathcal{A}^{(\text{geo})} + \mathcal{A}^{(\text{geo})} + \ldots + \mathcal{A}^{(\text{geo})})}_{2^d \text{ assembly tasks for the } 2^d \text{ cells adjacent to one vertex}} + \mathcal{S}^{(DoF)} \circ (\mathcal{A}^{(\text{geo})} + \ldots + \mathcal{A}^{(\text{geo})}) + \ldots \tag{4.3.1}$$

$\mathcal{S}^{DoF}$	Individual smoothing update of the multigrid algorithm

as a series of tasks over the grid entries, i.e. the unknowns. All operators are to be parameterised over the levels. As detailed in Section 3.3.2, our smoothers are applied element-wise, so we do not perform 2^d evaluations of each cell task $\mathcal{A}^{(\text{geo})}$. Instead, we set up the element matrix once per iteration, and immediately feed its impact on the surrounding u_ℓ values into the 2^d residuals. The smoother $\mathcal{S}^{(DoF)}$ then acts

on the residuals. An assembly of the intergrid transfer operators is omitted here, as for a wholly geometric implementation we know these operators and can hard-code them. This hard-coding is achieved by repeated evaluation of the $\mathcal{A}^{(\text{geo})}$ task. If the coarse grids are algebraic, such as our code's use of BoxMG, then a similar splitting into tasks must be constructed. However, we make no contribution there so do not detail such a splitting, see related work for a reference implementation of BoxMG on space-trees [33], [71].

This details what individual computation must be performed on each element and now we state how we implement these operations. We exclusively work with an element-wise assembly, where $\mathcal{A}^{(\text{geo})}$ computes the outcome of (3.1.2) over one cell c. Equation (4.3.1) has 2^d tasks—one task corresponding to each cell—that feed into the same smoother. Each $\mathcal{A}^{(\text{geo})}$ assembly task can in turn feed into 2^d $\mathcal{S}^{(DoF)}$ tasks—one for each vertex that is adjacent to the cell. It is convenient to evaluate each cell operator once, feed it the 2^d follow-up steps, and thus remove redundant tasks. We stress the entire procedure remains inherently additive however.

There are two methods of constructing coarse grid equations; they can be computed using rediscretisation—using geometric grid information—or algebraically—using fine grid equations. We neglect to specify the coarsening procedure here. Coarsening can be geometric or algebraic—our implementation uses geometric coarsening. We explain how both can be viewed as a collection of subtasks—similarly to the work we do on the fine grid.

With explicit geometric assembly, we run

$$(\mathcal{S} \circ \ldots \circ \mathcal{S}) \circ \left(\ldots + \mathcal{A}^{(\text{geo})}_{\ell-2} + \mathcal{A}^{(\text{geo})}_{\ell-1} + \mathcal{A}^{(\text{geo})}_{\ell} \right). \tag{4.3.2}$$

In (4.3.2), the task symbol $\mathcal{A}^{(\text{geo})}_{\ell}$ is a supertask bundling the evaluation of all the $\mathcal{A}^{(\text{geo})}$ tasks on level ℓ. This formalism relies on the insight that we can read multigrid cycles as iterations over one large equation system comprising of the fine grid and all coarse grid equations, if we commit to a generating system [95]. The addition in the assembly illustrates that submatrices within this large equation system can

be constructed (assembled) concurrently, as the equations within individual cells between levels are independent of each other.

With explicit algebraic assembly, we instead run

$$(\mathcal{S} \circ \ldots \circ \mathcal{S}) \circ \left(\ldots + \mathcal{A}^{(\text{alg})}_{\ell-2} + \mathcal{A}^{(\text{alg})}_{\ell-1} + \mathcal{A}^{(\text{geo})}_{\ell} \right). \tag{4.3.3}$$

$\mathcal{A}^{(\text{alg})}$	Algebraic coarse grid assembly process

$\mathcal{A}^{(\text{alg})}_{\ell-1}$ first constructs intergrid transfer prolongation $P^{\ell}_{\ell-1}$ and restriction $R^{\ell-1}_{\ell}$ between this level and next finest. A true algebraic assembly process would also construct the coarse grid space algebraically—we neglect this step and focus on geometric coarsening. We therefore introduce k, to represent the increase of the mesh size between levels; most literature uses $k = 2$ or $k = 3$. Our work—using the Peano framework [42]—uses the latter. $P^{\ell}_{\ell-1}$ and $R^{\ell-1}_{\ell}$ propagate a solution representation from the (coarse) mesh with spacing kh (level $\ell - 1$) to the (fine) grid with mesh size h (level ℓ) and back. Their construction—our work takes advantage of BoxMG [21], [33]—takes the fine grid operator's effective local null space into account. After that, $\mathcal{A}^{(\text{alg})}_{\ell-1}$ determines the matrix $A_{\ell-1} = R^{\ell-1}_{\ell} A_{\ell} P^{\ell}_{\ell-1}$ through the Ritz-Galerkin formulation. $\mathcal{A}_{\ell-1}$ in total yields three matrices $A_{\ell-1}$, $R^{\ell-1}_{\ell}$ and $P^{\ell}_{\ell-1}$. These definitions are recursive: $\mathcal{A}^{(\text{alg})}_{\ell-2}$ is similarly defined in terms of $\mathcal{A}^{(\text{alg})}_{\ell-1}$.

If coarse grid equations are defined recursively, there exists a partial ordering between grid levels—coarse grid equation construction tasks require relevant fine grid equation construction tasks to have terminated. Specifically, due to our use of space-trees, before a coarse grid equation can be computed for a coarse grid cell, fine grid equations must already be computed for all k^d child fine grid cells. This is a constraint that must be considered in the implementation of subtasks.

We return to our handling of the fine grid equations and more specifically define the subtasks.

Idea 1. *Exact integration over finite elements is expensive. We instead initially approximate this via a low accuracy numerical integration using piecewise constant*

quadratures of the material parameter ϵ. This approximation can subsequently be improved with higher accuracy integrations.

To make a finite element discretisation consistent, the assembly $\mathcal{A}^{(\text{geo})}$ has to evaluate (3.1.2) over all cells consistently, i.e. in the same way for all of a cell's adjacent vertices/stencils. This is trivial for constant ϵ, as we can extract weights and ϵ from the integral and integrate over the remaining shape functions analytically. That is, if we know ϵ within a vertex, we can precompute (3.1.2) for $\epsilon = 1$ and scale it on demand. If in (3.1.2) $\epsilon \neq const$, the computation is less straightforward and typically has to be computed numerically. For this, one option is to approximate ϵ. A polynomial approximation makes limited sense, as we are particularly interested in sharp ϵ transitions (material parameter jumps). Higher order polynomials would induce oscillations. We can, however, approximate ϵ as a series of constant values, i.e. we subdivide each cell into a Cartesian, equidistant subgrid with n^d volumes. Per volume, we assume ϵ to be constant.

For $n = 1$, such a subcell integration is equivalent to sampling ϵ once per cell centre (Fig. 4.3). Our ideas work with any numerical integration scheme acting upon the elements; however, we stick to our piecewise constant approximation of ϵ for simplicity. We stick to the simplest method of numerical integration as we do not contribute new ideas to the quality of numerical integrations. The numerical integration can be expensive—not due to the arithmetic load but due to the fact that ϵ lookups might be memory-access intense and thus slow—which strengthens the case for an element-wise realisation of the assembly, i.e. it is better to make $\mathcal{A}^{(\text{geo})}$ act per cell and feed into the 2^d adjacent vertices rather than computing (3.1.2) per $\mathcal{S}^{(DoF)}$ invocation. The latter option would effectively integrate (3.1.2) 2^d times.

If material parameters are exactly aligned with the cells, i.e. the jumps were aligned with cell faces, then a single sampling point per cell immediately gives an accurate stencil. However, the alignment of the cell geometry with the material parameter dictates the number of sampling points required to accurately capture the material

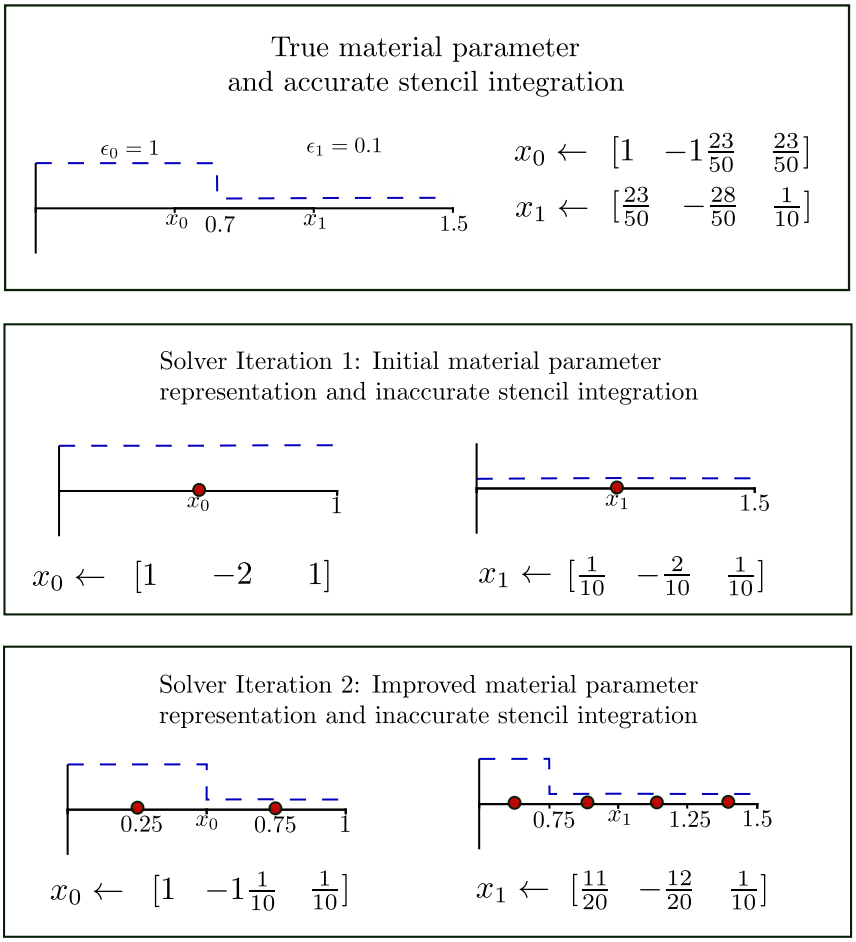

Figure 4.2: An example material parameter with discontinuity partway through a cell for a one-dimensional setup. The blue dotted line represents the material parameter and red dots sampling points. Top: The true material parameter and accurate stencils integrated for two vertices. Middle: Initial stencils used by a solver—one material point sampled per stencil giving inaccurate values. Bottom: Subsequent stencils used by a solver—differing number of points sampled per stencil.

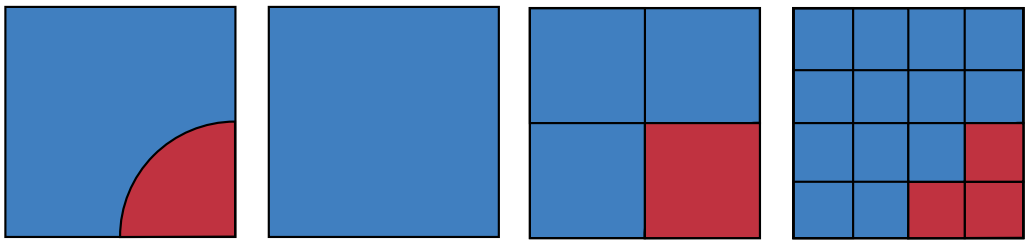

Figure 4.3: Exact material parameter within a cell (left) and a splitting of the material parameter into n^d quadrants for numerical integration (right).

parameter within the stencil. Rapidly changing material parameters require additional sampling points. We initially assume we can accurately represent the material parameter with a single sampling point—full well knowing this is may not be the case. An example discontinuous material parameter for a one-dimensional setup, and corresponding stencils for two vertices next to the discontinuity, is shown in Fig. 4.2. The first vertex, x_0, is at $x = 0.5$, and the second , x_1, at $x = 1$, while the discontinuity is over the point $x = 0.7$, i.e. not at a vertex. Consistent finite element stencils require consistent and accurate handling of the material parameter. Initial integrations only sample material parameters at the vertices. This is inaccurate—stencils are consistent but will give inaccurate solution values. Additional sampling points are used in subsequent iterations, vertex x_0 samples once per element centre and x_1 samples twice per element. This shows inconsistent representations due to the differing sampling points, but the increasing accuracy from using additional sampling points is clear.

$\mathcal{A}^{(\text{geo})}(n)$	Geometric assembly process for level ℓ with n sampling points
n	number of subcells divisions in each dimension

We know that different ϵ distributions require different choices of n, therefore it is convenient to parameterise the assembly tasks as $\mathcal{A}^{(\text{geo})}(n)$. A fast assembly—either explicit or embedded into the solves—requires the evaluation of $\mathcal{A}^{(\text{geo})}(n)$ to be fast; in particular as without explicit storage of the matrix, we evaluate each task once per cycle, i.e. multiple times overall. Therefore, for fixed n, it is in the interest of

the user to choose n as small as possible—$\mathcal{A}^{(\text{geo})}(n)$'s workload is in $\mathcal{O}(n^d)$—yet still reasonably accurate. Our current implementation averages the material parameter over each cell in a numerical integration of the weak formulation. Greater accuracy could be seen using a real homogenisation scheme, with the true Ritz-Galerkin operator, or adaptive quadrature shapes within the integration region, but is out of scope.

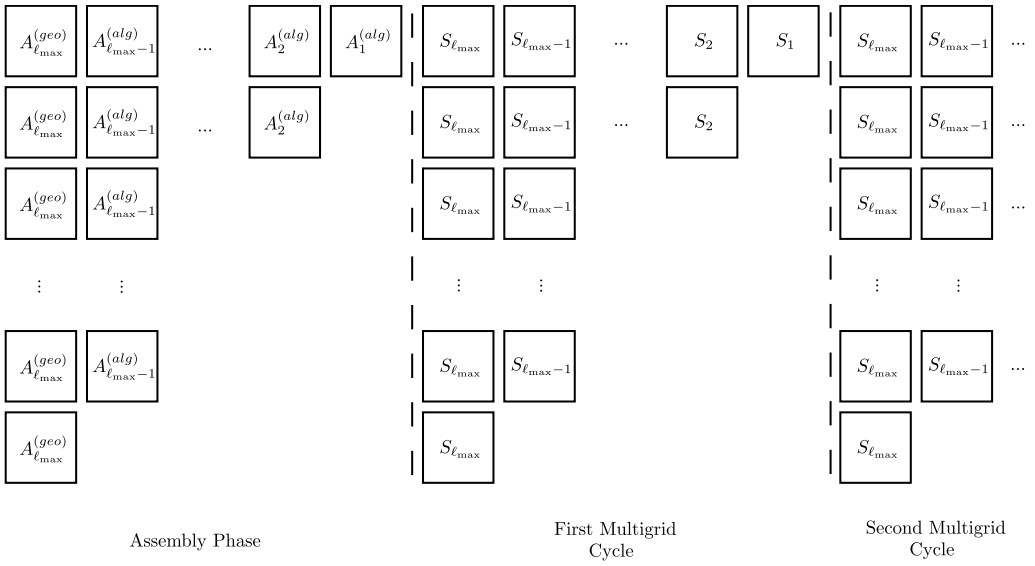

Figure 4.4: Task representation of early multigrid cycles with multilevel equation assembly performed a priori. Each box corresponds to a task. We perform an initial assembly phase, consisting of a series of level specific tasks. On level ℓ_{max} we perform the set of stencil construction tasks $A^{(geo)}_{\ell_{max}}$—one for each cell. We can then subsequently and sequentially compute coarse grid stencils $A^{(alg)}_{\ell_{max}-1}$ using Ritz-Galerkin. Post-assembly we start smoothing. We smooth all elements on the finest grid level initially (and in parallel)—this is the set of $S_{\ell_{max}}$ tasks. We can then smooth the coarse grid ℓ_{max-1} with the smaller set of smoothing operations, $S_{\ell_{max}-1}$, there. This recurses for additional coarse grids and repeats for subsequent smoothing steps.

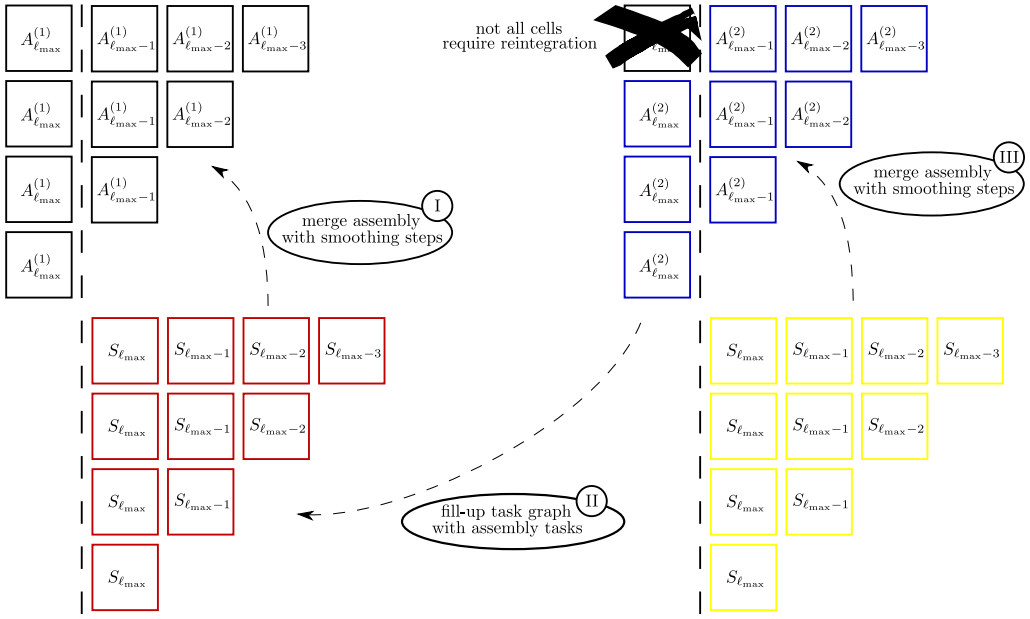

Figure 4.5: Construction of our delayed assembly. We use the same visual task representation and breakdown as in (Fig. 4.4). That is, a box corresponds to either an assembly or smoothing task acting upon a single cell. The stencil integration $A_{\ell_{\max}}$ is broken down into iterative substeps starting with a low-order approximation $A^{(1)}_{\ell_{\max}}$. We intermingle them with the earliest multigrid smoothing steps. Some stencils require further, more accurate integration $A^{(n)}_{\ell_{\max}}$. Each stencil update requires us to recompute the algebraic coarse grid operators.

4.4 Delayed stencil integration

The classic assembly phase (4.3.2) determines all multigrid operators prior to the (first) smoother application (Fig. 4.4). This is an expensive, initial step that delays the start of the solve. Here, we propose a means of eliminating this for a "lazy" alternative (Fig. 4.5). It is straightforward to implement a lazy implementation of the assembly along the lines of lazy evaluation in functional programming languages. Here, lazy denotes an on-demand evaluation of functions just before their result is required. In the present case, this means that the local assembly matrix is computed just prior to its first usage. The first time we access a cell during a smoother application step, we assemble the local element-wise matrix. An element-wise matrix is determined by either integrating (3.1.2) with an n^d subgrid (the fine grid), or

by computing the multigrid stencils plus the intergrid transfer operators due to the BoxMG/Ritz-Galerkin formulation (the coarse grid). The coarse grid assembly $\mathcal{A}_{\ell-1}$, consisting of both the construction of the element-wise operators plus intergrid transfer operator entries of $R_\ell^{\ell-1}$ and $P_{\ell-1}^\ell$, is thus by definition already a series of ready tasks. If a level ℓ is also a coarse grid then the equation on level $\ell-1$ is computed recursively.

Idea 2. *We do not perform an explicit, initial assembly phase. Instead we delay the computations and obtain the result once the smoother iterations begin.*

For multiplicative multigrid, this works naturally as we have a causal dependency between levels. We visit them from fine to coarse. Consequently, all level operators of level ℓ are available when we hit $\ell-1$ for the first time. They have no incoming, unresolved dependencies. They can be executed straight-away. This observation holds for both geometric and algebraic multigrid operator variants. For additive multigrid, on the other hand, this straightforward lazy stencil integration works if and only if we stick to rediscretisation and geometric transfer operators. It breaks down as we switch to algebraic operators, unless we prescribe the order that the levels are traversed, i.e. unless we ensure that the traversal of level ℓ is complete before we move to level $\ell-1$. We can weaken this statement [32], [42] and enforce that only those elements from level $\ell+1$ within the input of a chosen vertex's local P are ready. While this might be convenient for many codes, it eliminates some of additive multigrid's asynchronicity and thus one of its selling points.

With a lazy stencil integration, we can only utilise geometric coarse grid operators, unless we accept that an access to a coarse grid stencil can trigger the lazy (on-demand) evaluation of assembly steps on finer levels. We relax the assembly even further. We weaken the information flow constraints within the assembly or the accuracy demands on the fine grid operator. That is, we accept that fine grid operators stem from a low-accuracy integration, or that coarse grid operators do not yet hold appropriate Ritz-Galerkin data, even though we already use them.

Lazy evaluation then is a particular flavour of a delayed operator assembly, where missing information input is not tolerated but resolved on-demand. Lazy integration delays the assembly and, hence, the synchronisation, but still adds it when results are needed. Delayed integration in general, however, does not require us to wait for all input and thus does not stick to the precise mathematical rules. We drop synchronisation.

4.5 Adaptive stencil integration

Idea 3. *We minimise the number of subcells used in a quadrature on a per element basis. It is an adaptive integration.*

If a proper global choice of n for the fine grid (as well as for the rediscretisation if we stick to geometric operators) is not known a priori, we can employ an adaptive parameter selection:

Again, let $\mathcal{A}^{(\text{geo})}(n)$ denote the assembly of the local assembly matrix of one cell. For the evaluation of (3.1.2), it is parameterised by a suitable $n \in \mathbb{N}$, $1 < n < N$ i.e. by the numerical subsampling factor for the cell. Here N is a numerical subsampling factor that gives machine precision. That is, for an adaptive stencil integration, the current state of the integration per cell is determined by n. An initial low value for n generates a quick/cheap local element matrix that a cell can feed into the smoother. More accurate element matrices using successively larger values of n are computed, until an element matrix is deemed sufficiently accurate. These intermediary element matrices are used by the earlier smoothing updates: a smoother always uses the most accurate element matrix that it has access to. Different cells terminate this process independently. An effective termination criterion is when an element-wise norm for the local matrix is below a constant C.

Idea 4. *Early iterations of a smoother use less accurate matrix representations.*

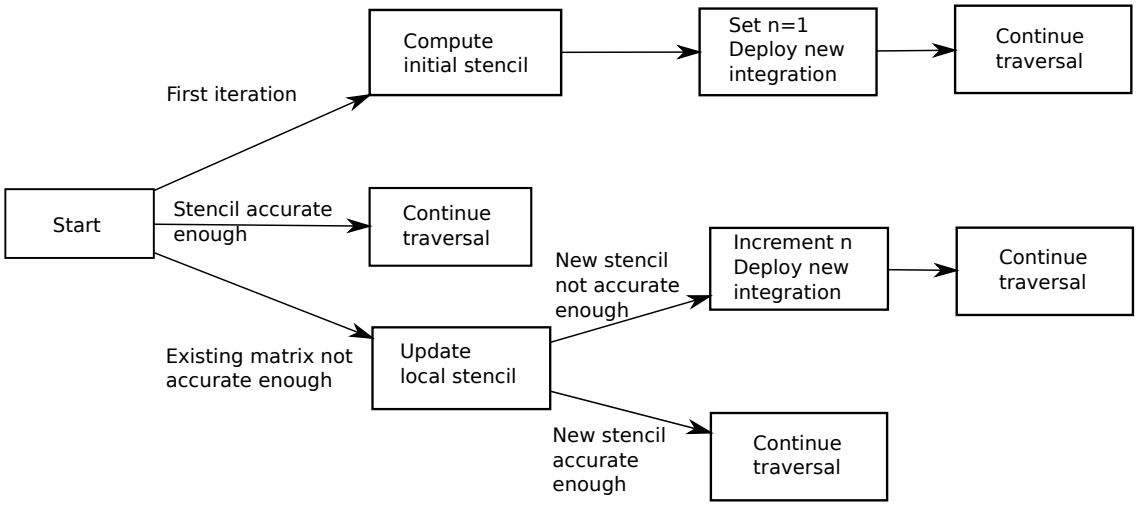

Figure 4.6: Illustrative diagram of how we perform the lazy integration. All cells carry a n that holds the number of samples per dimension of the quadrature.

Following the flow chart in Fig. 4.6 there are three state possibilities for a cell when adaptively integrating the element matrix:

1. There is no existing element-wise matrix. In this case, set the initial value of $n \leftarrow 1$ and evaluate (3.1.2) with a single sampling point in the centre of the cell. The resulting element matrix is stored to be used for subsequent calculations until future tasks for this task are evaluated.
2. The existing element-wise matrix is accurate enough. We continue to work with it.
3. There is an existing element matrix but it is of insufficient accuracy. The task evaluates (3.1.2) over the cell of interest. It uses a $(n+1)^d$ subgrid to discretise ϵ. This new matrix is $A^{(\text{new})}$. The previous matrix $A^{(\text{old})}$ is then compared to $A^{(\text{new})}$ to determine if the integration is of sufficient accuracy. The sequence terminates if
$$\frac{\|A^{(\text{new})} - A^{(\text{old})}\|}{\|A^{(\text{old})}\|} < C.$$
for constant C. The choice of C should be in line with discretisation accuracy, i.e. $A^{(\text{new})}u \approx A^{(\text{old})}u$ to within $\mathcal{O}(h)$ for our setup. For simple setups a suitable C can be chosen empirically. If this does not hold then n is incremented and

future updates are computed when this cell is next accessed.

It is obvious that parameterisations of $P_\ell^{\ell+1}$ and $R_{\ell+1}^\ell$ make limited sense. However, BoxMG and algebraic methods make operators depend directly on the operator on level $\ell + 1$. This dependency propagates all the way through to the fine grid. Therefore, both P and R will depend indirectly on the n choice of the algorithm for algebraic implementations.

We assume that $n \leftarrow n + 1$ for successive integrations. Therefore, after at most $n_{\max} + (\ell_{\max} - 1)$ steps all local equation systems are valid—as long as the grid is stationary, and we tackle a linear problem. $n_{\max}$ is the maximum integration accuracy over all cells that is required eventually. It is not known a priori. The subsequent $(\ell_{\max} - 1)$ correspond to the construction of the $(\ell_{\max} - 1)$ Ritz-Galerkin coarse grid equations. More aggressive incrementing of n can reduce the total number of iterations, but increases the cost of each integration and can introduce redundant integrations.

The scheme describes an adaptive quadrature rule, where the accuracy of the integrator is cell-dependent and determined by an iterative process. This iterative process terminates as soon as a further increase of the accuracy does not yield significantly improved stencils anymore.

4.6 Asynchronous and anarchic stencil integration

With the iterative scheme at hand, it is straightforward to construct an asynchronous stencil integration, where the actual integration is deployed to a task of its own and runs in parallel to the solver's iterations (Fig. 4.5). We simply deploy the previously defined tasks in parallel with existing work and use the updated matrices as and when they drop in. The specifics of how this is implemented is covered in Chapter 6.

Algorithm 5 Adaptive integration algorithm. Called when a cell is encountererd for the first time during a grid traversal. n is the number of sampling points for numerical integration. IntegrateStencil(n) represents the act of the numerical integration up to accuracy n.

```
function greedy-integration
  if First iteration then
    n ← 1
    IntegrateStencil(n)
    Wait until initial integration terminates
    LocalStencil ← UpdatedStencil
    n ← 2
    IntegrateStencil(n)
  else if Stencil sufficiently accurate then
    Take no action
  else
    if IntegrationTerminated then
      if UpdatedStencil sufficiently similar to LocalStencil then
        Stencil is sufficiently accurate
      else
        n ← n + 1
        IntegrateStencil(n)
      end if
      LocalStencil ← UpdatedStencil
      IntegrationTerminated ← False
    end if
  end if
end function
```

Idea 5. *We do not require synchronisation between multigrid iterations and adaptive integration iterations. They are performed independently.*

This is shown in Alg. 5. The key difference between the synchronous and asynchronous versions of the algorithm is the check "integration terminated". In the synchronous version, the integrations are performed immediately and the result stored. There is no need to check if they've been performed. Obviously this is not the case when they're computed in parallel.

With an anarchic stencil integration, we have no control over when and with which integration accuracy we use. In either case, fine grid stencil integrations are deployed to the background and once they drop in, all affected coarse grid operators become invalid in a Ritz-Galerkin sense.

It is obvious that the iterative, delayed stencil integration can be applied to all levels if we stick to rediscretisation. With algebraic operators, the technique applies only to the finest grid level. If we use iterative stencil integration on coarser levels, we have to control the termination criterion in (3) carefully: As the coarse equations are only correction equations and are "only" solved up to a mesh-dependent accuracy in classic multigrid terminology, it makes limited sense to choose the threshold C there uniformly and small on all resolution levels, i.e. C is chosen dependent upon the accuracy of the mesh/discretisation.

4.7 Vertical rippling

When working in a Ritz-Galerkin environment, possibly also with BoxMG, we can either deploy the computation of the three arising coarse operators to background tasks, too, or we can recompute these operators in each and every cycle. Given the limited and deterministic computational load, the latter might be reasonable.

Idea 6. *We give up on the idea of vertical synchronisation per multigrid iteration. We use outdated coarse grid operators in multigrid cycles when the fine grid equations change and update them in later iterations.*

If the operators, however, are determined in the background, we can spawn only those tasks that might actually yield changed operators. An analysed tree grammar [103] formalises the requirements: If a matrix update changes the stencil associated with a vertex v_ℓ which in turn is in the image of a stencil $P^\ell_{\ell-1}$ associated with a vertex $v_{\ell-1}$, then the 2^d adjacent cells of $v_{\ell-1}$ should be flagged. In the next multigrid cycle, all flagged cells' $A_{\ell-1}, P, R$ computations should be repeated, taking the new fine grid operator A_ℓ into account. The same level-by-level information propagation—shown in Fig. 4.7—formalises how information propagates through both space and mesh resolutions, if we recompute all three operators in each and every multigrid cycle.

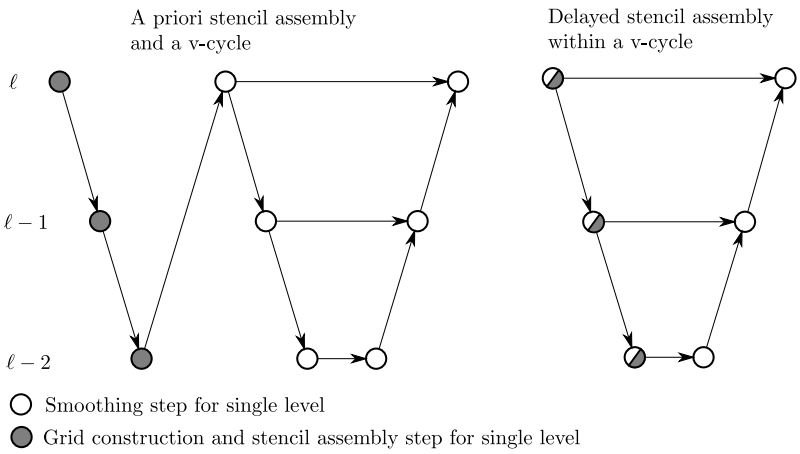

Figure 4.7: Diagrammatic view of computing coarse grid equations prior to a solver iteration (left) compared to plugging into a grid traversal of the actual solver.

If we employ dynamically adaptive mesh refinement, the mesh coarsening or refinement induces changes of operators. As a result, they implicitly trigger coarse grid operator updates. A similar argument holds for nonlinear setups: If the nonlinear component induces signification changes in the fine grid operator—for many operators, this might be a localised effect, i.e. the fine grid equation system might not change everywhere—we have to change a set of affected coarse grid operators.

In an additive setting, delayed operator updates ripple through the equation system, i.e. the updates propagate upwards by at most one grid level per cycle. Coarse grid operators on a level ℓ lag behind the fine grid operators on level $\ell_{\max}$ by $\ell_{\max} - \ell$ iterations.

4.8 Full multigrid cycles and dynamic adaptivity

Full multigrid, and any solver that employs adaptive mesh refinement for that matter, undergo repeated explicit assembly phases due to the coarse grid operator updates. The quality of the discretisation therefore improves after these assembly phases, but the overall cost of assembly is worsened. Our adaptive integration reduces this cost.

Full multigrid, extends the fine-to-coarse idea of multigrid with a coarse-to-fine

flavour: We start with a rather coarse fine grid mesh and run our multigrid solve there. This initial mesh then is "unfolded" into the next finer resolution and we continue. As the unfolding is combined with a (higher order) prolongation—the higher order can be dropped if we are willing to invest additional smoothing steps—a solve on a level $\ell - 1$ serves as initial guess to the solve on level ℓ. We solve

$$\ldots \circ \mathcal{A}_\ell \circ \mathcal{A}_{\ell+1} \circ \mathcal{A}^{(\text{geo})}_{\ell+2}(\epsilon) \circ \mathcal{U} \circ \mathcal{S} \circ \mathcal{A}_\ell \circ \mathcal{A}^{(\text{geo})}_{\ell+1}(\epsilon) \circ \mathcal{U} \circ \mathcal{S} \circ \mathcal{A}^{(\text{geo})}_{\ell}(\epsilon)$$

$\mathcal{U}$	mesh refinement operator

where $\mathcal{U}$ is the mesh unfolding operator. The term "unfolding" technically describes mesh refinement. Whenever we equip our multigrid solver with dynamic adaptivity, i.e. the solver may add more degrees of freedom to the mesh throughout the solve, we technically inject (localised) $\mathcal{U}$ operators into the mesh and must trigger some reassembly. From an implementation point of view, full multigrid cycles and dynamic adaptivity share common properties. All properties, including full multigrid mesh unfolding, thus hold for additive and multiplicative as well as both geometric and algebraic multigrid.

This iterative unfolding of the mesh triggers an assembly phase each time any component in the mesh changes. In multigrid, this is not just a local update, but requires updates to all levels in the mesh hierarchy. Vertical rippling eliminates the assembly phase as a discrete step, instead intermingling the assembly phase with the solve phase. Coarse grid equations are assumed not to change dramatically after a refinement. The old matrix equations are treated as suitable approximations of the new and used in coarse grid smoothing steps until the recomputed coarse grid equations "ripple" up to that level. This effectively eliminates the synchronisation of grid equations across levels after a refinement and reduces the delay before solution updates are produced again after a refinement.

4.9 Incorporating other/non-Jacobi smoothers

At first glance, there is no apparent link between the choice of smoother and our delayed method of stencil assembly. Any choice of smoother would appear to be equally viable, as from an implementation perspective any existing smoother could take advantage of delayed asynchronous assembly the same way it would an a priori assembly. From the solver's point of view, it merely accesses a representation of the local operator and constructs the smoother using that data. The delayed assembly process would provide local operators through the same interface, but the operators would be constructed totally differently in the backend. This pattern of access would work well with a smoother such as Red-Black Gauss-Seidel, which is already highly parallel. However, upon closer inspection there may be an underlying connection that must be taken into account during the implementation.

Our current smoother of choice is a simple Jacobi smoother. This is slow to converge, and therefore unlikely to be impacted by errors in the representation of a local operator. The final, accurate, integration is more likely to be ready by the time a Jacobi smoother has converged to a final solution than with a smoother that converges more quickly. Jacobi is a simple smoother—it only uses the diagonal of a matrix equation being solved—so is also more forgiving of errors in the integration operator. Smoothers that converge faster, such as Red-Black Gauss-Seidel or Successive Over Relaxation, could converge to a solution that corresponds to an incorrect operator representation, due to the final integration not being assembled in time. This would have to be accounted for in the implementation. For example, higher priorities could be assigned to integration tasks, so that they're completed in earlier iterations or a more aggressive adaptive scheme could be employed so fewer numerical integrations iterations are required. Furthermore, a more powerful smoother is more sensitive to errors in the representation of the operator than a less powerful one. A line smoother, for example, acts on an entire face within the mesh, and solves for that full face. This may be severely impacted by inaccurate solution representation over

the face. Inconsistent representations between fine grid stencils may result in a matrix that is singular, such smoothers are more sensitive to errors in the operator representation. Therefore additional checks may be required on stencil quality—so that local matrices do not become singular and also produce solutions of reasonable quality.

4.10 Relationship to other notions of asynchronicity

Another scheme that introduces asynchronous principles to multigrid is the work of Chow and Wolfson-Pou [90], as discussed in Section 3.2.6. Their asynchronous multigrid uses the idea of asynchronous smoothers—smoothing is performed by smoother threads that act totally independently of other smoother steps/threads. These smoothers may produce corrections at differing rates, that is, they become out of sync. Certain smoother threads, and the corresponding topological regions or grid levels, may therefore produce updates less frequently than others, or operate on "outdated" solution information. Asynchronous multigrid algorithms are built on top of additive multigrid—the alternative, multiplicative multigrid, would require synchronisation between grid levels for pre- and postsmoothing, which is at odds with the asynchronous principle.

Our work uses two notions of asynchronicity that differ from theirs: asynchronous grid construction, and asynchronous processing of grids. We use an asynchronous method of grid construction, rather than smoother application—updated forms of the fine grid equations drop in anarchically but all corrections from smoothers are implicitly synchronised between correction steps. Therefore, adaptive stencil integration/asynchronous grid construction is another flavour of asynchronous ideas. It is orthogonal to asynchronous multigrid. As we use an element-wise smoother application, this would allow corrections to be computed asynchronously too; however,

the application of corrections and restriction of residuals to coarse equations forces a degree of synchronisation. Chow and Wolfson-Pou refer to this application of asynchronous ideas as "asynchronous task-based processing of grids" and emphasise that it differs from their main application of asynchronicity.

Both our asynchronous ideas and theirs can be seen to produce "inaccurate" residuals. Our residuals do not use synchronised coarse grid equations and intergrid transfer operators—residuals that coarse grids work with are not "true" residuals. The work of Chow and Wolfson-Pou may produce inaccurate residuals due to smoothing steps not being synchronised—subpartitions may not receive corrections from other subpartitions. In this case, local corrections will not enter the right-hand side of other correction equations.

There is no fundamental reason why lazy and adaptive stencil integration and asynchronous multigrid could not be used within the same solver. Asynchronous smoothers create threads that smooth equations independently and our adaptive integration tasks could readily feed updated stencils into those threads. Smoothers could independently handle the delayed construction and iterative improvement of local matrix equations. Combining these two techniques would negate another synchronisation step within multigrid. Chow and Wolfson-Pou have also observed improved robustness due to the introduction of asynchronicity. Our techniques could see similar improvements if we merged both ideas.

Chapter 5

Additive Damping Scheme

In the previous chapter we covered how we improve time-to-solution of a multigrid solver by pipelining the assembly and thus negating some of the algorithmic latency. Assembly is only part of the story. Overall time-to-solution is also effected by the multigrid cycles themselves. We provide further improvements through the multigrid cycles—we introduce a damping parameter that stabilises additive multigrid to reduce the overall number of iterations required for convergence. The construction and theory behind that damping parameter is the focus of this chapter. We start by giving an outlook on why additive multigrid exhibits instabilities. Then throughout the chapter build up to an overview of the theory behind our choice of damping parameter. We outline some choices that govern the implementation here but do not cover our own implementation—that is the subject of Chapter 6.

5.1 An additive multigrid solver

The generalised matrix representation of additive multigrid (3.2.1) reads as

$$u_{\ell_{\max}} \leftarrow u_{\ell_{\max}} + \left(\sum_{\ell=\ell_{\min}}^{\ell_{\max}} \omega_{\text{add}}(\ell) P_{\ell}^{\ell_{\max}} M_{\ell}^{-1} R_{\ell_{\max}}^{\ell} \right) (b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{\max}}) \,,$$

The following chapter is modified from text that was previously published in [1]—"Stabilised Asynchronous Fast Adaptive Composite Multigrid using Additive Damping". The text has been expanded throughout and the Section 5.4 and Section 5.7 are new.

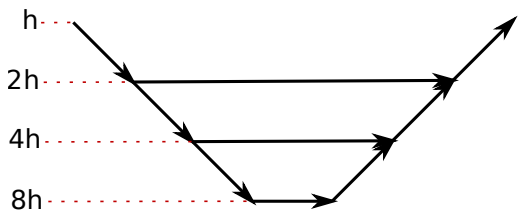

Figure 5.1: Representation of an Additive "V-Cycle" (transfer of data between grids). The residual is computed on the finest grid then this same residual is restricted to all grid levels.

where M_ℓ is an approximation to A_ℓ. We use the Jacobi smoother $M_\ell^{-1} = \text{diag}^{-1}(A_\ell)$ on all grid levels ℓ. No alternative (direct) solver or update scheme is employed on any level. The prolongation symbol $P_\ell^{\ell_{\max}}$ takes the solution on a particular level ℓ and projects it onto the finest level $\ell_{\max}$. The restriction symbol $R_{\ell_{\max}}^{\ell}$ works in the opposite direction and is usually the transpose of $P_\ell^{\ell_{\max}}$. In practice, we construct this operator from repeated application of the previously introduced $P_\ell^{\ell+1}$ transfer operators. Although the notation implies this is a single transfer between levels, it is performed a single grid level at a time. Restriction, again, works the other way round, i.e. projects from finer to coarser meshes. Ritz-Galerkin multigrid [13] finally yields $A_\ell = R_{\ell+1}^{\ell} A_{\ell+1} P_\ell^{\ell+1}$ for $\ell < \ell_{max}$.

For an ℓ-independent, constant $\omega_{\text{add}}(\ell) \in (0, 1]$, additive multigrid tends to become unstable once $\ell_{\max} - \ell_{min}$ becomes large [14], [15], [32]: If the fine grid residual $b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{\max}}$ is homogeneously distributed then there are no high frequency errors for the fine grid smoother to eliminate. Instead it will attempt to eliminate low frequency error modii that the coarse grid is better suited at eliminating—effectively attempting to reduce the same error multiple times. The coarse grid operator therefore pushes the solution in the same direction as the fine. Summation of all level contributions therefore over-relaxes and moves the solution too aggressively in this direction. This effectively removes "too much" of the error. If the residuals are not homogeneously distributed (fine grid error is not smooth), the restricted residuals from the fine grid would "average out" the high frequency detail. The fine grid and the coarse grid would therefore be solving for different error modii.

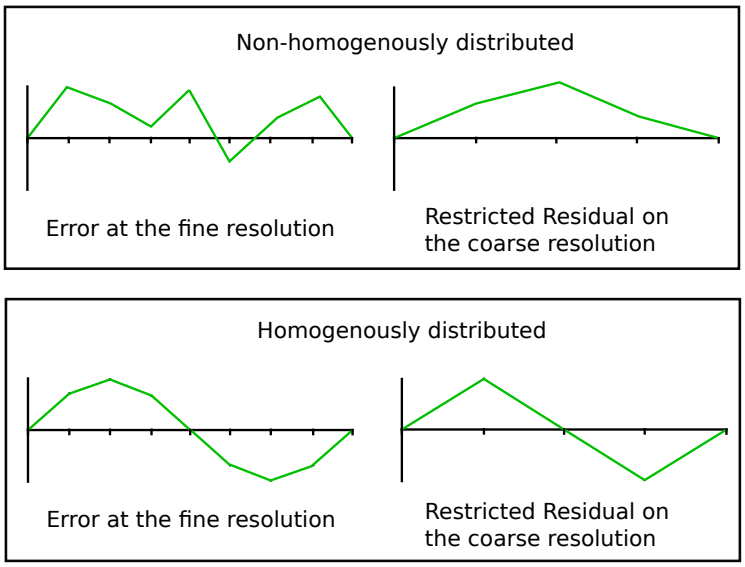

Figure 5.2: Top: Non-homogeneously distributed error on the fine grid. The fine detail is not apparent on the coarse. Bottom: Homogeneously distributed error on the fine grid. When represented on the coarse it captures the same detail.

This is illustrated in Fig. 5.2—where a comparison is shown between homogeneous and non-homogeneous fine grid errors. Homogeneous fine grid error distributions can accurately be represented on a coarse grid, but this is not the case for non-homogeneous distributions where the fine detail cannot always be captured. A straightforward fix to overcorrection is exponential damping $\omega_{\text{add}}(\ell) = \hat{\omega}_{\text{add}}^{\ell_{\max}-\ell}$ with a fixed $\hat{\omega}_{\text{add}} \in (0, 1)$. Here the exponent corresponds to an actual exponent—not a superscript. If an adaptive mesh is used, $\ell_{\max} - \ell$ is ill-suited as there is no global $\ell_{\max}$ hosting the solution. This is illustrated in the bottom right of Fig. 5.1. Such exponential damping, while robust, struggles to track global solution effects efficiently once many mesh levels are used: The coarsest levels make close to no contribution to the solution. Previous work by Reps and Weinzierl [32] instead made $\ell_{\max}$ a per-vertex property—therefore the relaxation parameter on one level is position dependent. A per-vertex damping parameter also means an additive scheme is able to both handle a changing number of mesh levels more readily and larger jumps in a changing material parameter. Their damping parameter is derived from

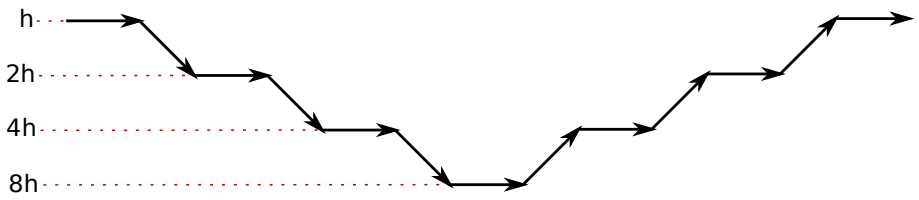

Figure 5.3: Representation of possible errors removed via an multiplicative smoothing cycle with presmoothing and postsmoothing steps. Presmoothing prevents the next coarser level from producing corrections for the same error as the finer level. Postsmoothing prevents projected corrections from introducing new errors on the fine grid.

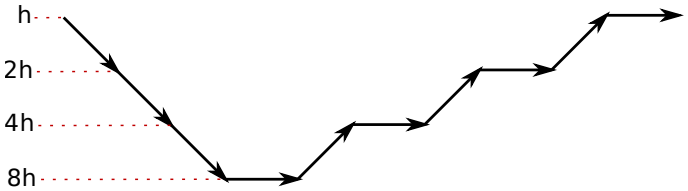

Figure 5.4: Representation of a multiplicative V-Cycle with no presmoothing steps. The finest grid smooths the error which is then restricted to the coarsest grid level and sequentially smoothed on increasingly finer grids.

a tree grammar [103].

Multiplicative multigrid is more robust than additive multigrid by construction. Multiplicative multigrid does not make one residual feed into all level updates in one rush, but updates the levels one after another (see Fig. 5.3). It starts with the finest level. Before it transitions from a fine level to the next coarsest level, it runs some approximate solves (smoothing steps) on the current level to yield a new residual. We may assume that the error represented by this residual is smooth. Yet, the representation becomes rough again on the next level, where we become able to smooth it efficiently again. Cascades of smoothers act on cascades of frequency bands. Multiplicative methods are characterised by the number of the pre- and postsmoother steps μ_{pre} and μ_{post}, i.e. the number of relaxation steps before we move to the next coarser level (pre) or next finer level (post), respectively.

We restrict our focus here to additive schemes which only use a single smoothing step

per-level—this follows from a review of additive and multiplicative multigrid [14], where additional smoothing steps are not shown to provide a clear improvement to the rate of convergence. The multiplicative multigrid solve closest to an additive scheme with only one smoothing application is a $V(0, \mu_{post})$-cycle, i.e. a scheme without any presmoothing and μ_{post} postsmoothing steps (see Fig. 5.4). Different to additive multigrid, in a $V(0, \mu_{post})$-cycle, the effect of smoothing on a level ℓ here does feed into the subsequent smoothing on $\ell + 1$. Since $\mu_{pre} = 0$ yields no classic multiplicative scheme—the resulting solver does not smooth prior to the coarsening—we conclude that the $V(\mu_{pre} = 1, 0)$-cycle thus is the (robust) multiplicative scheme most similar to an additive scheme. The multiplicative $V(1, 0)$ two-grid scheme with exact coarse grid solve reads

$$\begin{aligned} u_\ell \;\leftarrow\; & P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1} (b_\ell - A_\ell \left[u_\ell + \omega_\ell M_\ell^{-1} (b_\ell - A_\ell u_\ell) \right]) \\ & + \left[u_\ell + \omega_\ell M_\ell^{-1} (b_\ell - A_\ell u_\ell) \right]. \end{aligned} \tag{5.1.1}$$

Next, we compare this multiplicative representation directly to the additive representation.

5.2 An additively damped additive multigrid solver

Both additive and multiplicative multigrid sum up all the levels' corrections. Multiplicative multigrid is more stable than additive—it does not overshoot—as each level eliminates error modes tied to its resolution before other levels begin their respective resolution and tackle their error modes. In practice, we cannot totally separate error modes, and we cannot assume that a correction on level ℓ does not introduce a new error on level $\ell + 1$. Multigrid solvers thus often use postsmoothing. Once we ignore this multiplicative lesson, the simplest class of multiplicative solvers is $V(\mu_{pre} = 1, 0)$.

We start with a recast of the multiplicative $V(1,0)$ two-grid cycle (5.1.1) into an additive formulation. Our objective is to quantify additive multigrid's over-correction relative to its multiplicative cousin. For this, we compare the multiplicative two-grid scheme, denoted $u_{\ell,\text{mult}}$, to the two level additive scheme with an exact solve on the coarse level

$$u_{\ell,\text{add}}^{(n+1)} = P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell u_\ell^{(n)}) + \left[u_\ell^{(n)} + \omega_\ell M_\ell^{-1}(b_\ell - A_\ell u_\ell^{(n)})\right].$$

The difference is

$$\begin{aligned} u_{\ell,\text{mult}}^{(n+1)} - u_{\ell,\text{add}}^{(n+1)} &= P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell \left[u_\ell^{(n)} + \omega_\ell M_\ell^{-1}(b_\ell - A_\ell u_\ell^{(n)})\right]) \\ &\quad - P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell u_\ell^{(n)}) \\ &= -P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1} A_\ell \omega_\ell M_\ell^{-1}(b_\ell - A_\ell u_\ell^{(n)}). \end{aligned} \tag{5.2.1}$$

The superscripts $^{(n)}$ denotes old iterates of a vector, whereas $^{(n+1)}$ denotes new iterates respectively. We continue to omit it from here where possible.

Starting from the additive rewrite of the $V(1,0)$ multiplicative two-level scheme, we express multiplicative multigrid as an additive scheme. Additive multigrid tends to more readily show improved performance on a large scale parallel implementation than multiplicative multigrid. There is no close-to-serial coarse grid solve. There is no coarse grid bottleneck in an Amdahl sense. Multiplicative multigrid, however, tends to converge faster and is more robust. Different to existing approaches such as mult-additive, our approach does not aim to achieve the exact convergence rate of multiplicative multigrid. Instead, we aim to mimic the robustness of multiplicative multigrid in an additive regime—i. e. allow additive multigrid to successfully converge across a wider range of setups. Our agenda starts from one main idea:

Idea 7. *We add an additional one-level term to our additive scheme which compensates for additives overly aggressive updates compared to multiplicative $V(1,0)$ multigrid.*

This idea describes the rationale behind (5.2.1), where we stick to a two-grid formalism. Our strategy next is to find an approximation to

$$-P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1} A_{\ell} \omega_{\ell} M_{\ell}^{-1} (b_{\ell} - A_{\ell} u_{\ell}) \tag{5.2.2}$$

from (5.2.1) such that we obtain a modified additive two-grid scheme which, on the one hand, mimics multiplicative stability and, on the other hand, is cheap. For this, we read the difference term as an auxiliary solve.

Idea 8. *We approximate the auxiliary term (5.2.2) with a single smoothing step.*

The approach yields a per-level correction

$$-P_{\ell-1}^{\ell} \omega_{\ell-1} \tilde{M}_{\ell-1}^{-1} R_{\ell}^{\ell-1} A_{\ell} \omega_{\ell} M_{\ell}^{-1} (b_{\ell} - A_{\ell} u_{\ell}). \tag{5.2.3}$$

We use the tilde to denote the auxiliary solves. Following on from Idea 7, this is a per-level correction: When we re-generalise the scheme from two grids to multigrid (by a recursive expansion of $A_{\ell-1}^{-1}$ within the original additive formulation), we do not further expand the correction (5.2.2) or (5.2.3). The recursive expansion is the key idea behind multigrid and where it gets its power—we ignore it for the damping term. This implies another error, which we accept in return for a simplistic correction term without additional synchronisation or data flow between levels.

Idea 9. *The damping runs asynchronously to the actual solve. It is another additive term computed concurrently to each correction equation.*

Using $A_{\ell} \omega_{\ell} M_{\ell}^{-1}$ adds a sequential ingredient to the damping term. A fine grid solve must be finished before it can enter the auxiliary equation. This reduces concurrency. Therefore, we propose to merge this preamble smoothing step into the restriction. This is similar to smoothed aggregation which typically uses a simple aggregation/restriction operator, and then improves it by applying a smoother. Rather than enriching a simple tentative transfer operator, we differ and instead use a more powerful initial transfer and are merely interested in anticipating an additional fine

grid smoothing step. The idea is also similar to mult-additive [89], which constructs intergrid transfer operators that pick up multiplicative pre- or postsmoothing behaviour. Mult-additive applies this to both restriction and prolongation operators to the coarse grid level—we instead only apply the idea to a single operator to an auxiliary grid level. We apply the smoothed operator concept to the restriction $\tilde{R}_\ell^{\ell-1} = \omega R_\ell^{\ell-1} A M_\ell^{-1}$, and end up with a wholly additive correction term

$$- \tilde{\omega} P_{\ell-1}^{\ell} \tilde{M}_{\ell-1}^{-1} \tilde{R}_\ell^{\ell-1}, \tag{5.2.4}$$

with its own damping weight, $\tilde{\omega}$. This additional asynchronity is where we diverge from the path set by AFACx—the auxiliary grid space in AFACx must be handled prior to the original correction grid space it was built over. We merge the additional smoothing step with the restriction and therefore do not impose this requirement. Within our scheme, all grid levels—both auxiliary and original correction levels—can be processed in arbitrary order.

In algebraic schemes coarse grid identification and set up is a significant part of the overall assembly process and delays time-to-solution. The coarse grids also introduce additional memory overhead that can limit the performance of a solver. We effectively have to construct two coarse grid hierarchies, due to the addition of our auxiliary coarse grids, which could introduce redundant work if we explicitly construct two hierarchies. Minimising the costs of coarse grid assembly motivates our next idea.

Idea 10. *We geometrically identify the auxiliary coarse grid levels with the actual multilevel grid hierarchy. All resolution levels can integrate into a single space-tree.*

$\tilde{M}_\ell$ and $\tilde{A}_\ell$ are auxiliary operators but act on mesh levels which we hold anyway. With a defined, multilevel hierarchy we can rewrite the exact matrix inversion in (5.2.1) as a multilevel smoother iteration with each level incorporating its own additional damping term.

We therefore unfold the two-grid scheme into

$$u_{\ell_{\max}} \leftarrow u_{\ell_{\max}} + \left(\sum_{\ell=\ell_{\min}}^{\ell_{\max}} \omega_{\text{add}}(\ell) P_{\ell}^{\ell_{\max}} M_{\ell}^{-1} R_{\ell_{\max}}^{\ell} \right) (b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{\max}}) - \left(\sum_{\ell=\ell_{\min}}^{\ell_{\max}} \tilde{\omega}_{\text{add}}(\ell) P_{\ell}^{\ell_{\max}} \tilde{M}_{\ell}^{-1} \tilde{R}_{\ell_{\max}}^{\ell} \right) (b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{\max}}) \,, \quad (5.2.5)$$

where we set, without loss of generality, $M_{\ell_{\min}}^{-1} = 0$. This assumes that no level coarser than $\ell_{\min}$ hosts any degree of freedom.

Algorithms in standard AFAC literature present all levels as correction levels. That is, a global residual is computed on the composite grid and then restricted to construct the right-hand side of error equations on all grid resolutions. This includes the finest grid level. We instead use standard multigrid convention and directly smooth the finest grid level (Algorithms 6 and 7). We only restrict the residual to coarse grid levels.

These four ideas mean we see three key benefits: We stick to a geometric grid hierarchy and then also reuse this hierarchy for additional equation terms. We stick to an additive paradigm and then also make additional equation terms additive. We stick to a geometric-algebraic mindset.

5.3 Three damping operator choices

It is obvious that the effectiveness of the approach depends on an efficient and accurate approximation of the inverse within (5.2.4). We propose three variants. All three are based upon the assumption that smoothed intergrid transfer operators yield better operators than standard bi- and trilinear operators (and obviously naive injection or piecewise constant interpolation) [104]–[106]. Simple geometric transfer operators fail to capture complex solution behaviour [107]–[109] for non-trivial ϵ choices in (1.1).

For the three variants we introduce two modified, i.e. smoothed, intergrid transfer

Algorithm 6 Blueprint of one cycle of an adAFAC-Jac iteration without AMR. $R^{\ell}_{\ell_{\max}}$ or $P^{\ell_{\max}}_{\ell}$ denote the respective restriction or prolongation, between finest grid level and an arbitrary grid level. $\widetilde{R}^{\ell}_{\ell_{\max}}$ is the application of $R^{\ell+1}_{\ell_{\max}}$, followed by an application of a smoothed single level intergrid transfer operator.

function adAFAC-Jac
  $r_{\ell_{\max}} \leftarrow b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{max}}$
  for all $\ell_{\min} \leq \ell < \ell_{\max}$ **do** ▷ Restrict fine grid residual to grid levels ℓ
    $b_{\ell} \leftarrow R^{\ell}_{\ell_{\max}} r_{\ell_{\max}}$
    $\widetilde{b}_{\ell} \leftarrow \widetilde{R}^{\ell}_{\ell_{\max}} r_{\ell_{\max}}$ ▷ Auxiliary residual restriction
  end for
  for all $\ell_{\min} < \ell \leq \ell_{\max}$ **do**
    $c_{\ell} \leftarrow 0$; $\widetilde{c}_{\ell-1} \leftarrow 0$ ▷ Initial "guess" of correction and damping
    $c_{\ell} \leftarrow \text{JACOBI}(A_{\ell} c_{\ell} = b_{\ell}, \omega)$ ▷ Iterate of correction equation stored in c_{ℓ}
    $\widetilde{c}_{\ell-1} \leftarrow \text{JACOBI}(A_{\ell-1} \widetilde{c}_{\ell-1} = \widetilde{b}_{\ell-1}, \widetilde{\omega})$ ▷ Iterate of coarser damping equation
  end for
  $c_{\ell_{\min}} \leftarrow 0$; ▷ Initial "guess" on coarsest level
  $c_{\ell_{\min}} \leftarrow \text{JACOBI}(A_{\ell_{\min}} c_{\ell_{\min}} = b_{\ell_{\min}}, \omega)$ ▷ Iterate of correction equation
  $u_{\ell_{\max}} \leftarrow u_{\ell_{\max}} + c_{\ell_{\min}} + \sum_{\ell=\ell_{\min}+1}^{\ell_{\max}} P^{\ell_{\max}}_{\ell} c_{\ell} - P^{\ell_{\max}}_{\ell-1} \widetilde{c}_{\ell-1}$
end function

Algorithm 7 Blueprint of our adAFAC-PI without AMR. R^i or P^i denote the recursive application of the single level restriction or prolongation, R or P, respectively. I is the injection operator.

function adAFAC-PI
  $r_{\ell_{\max}} \leftarrow b_{\ell_{\max}} - A_{\ell_{\max}} u_{\ell_{max}}$
  for all $\ell_{\min} \leq \ell < \ell_{\max}$ **do**
    $b_{\ell} \leftarrow R^{\ell}_{\ell_{\max}} r_{\ell_{\max}}$ ▷ Restrict fine grid residual to coarser levels
  end for
  for all $\ell_{\min} < \ell \leq \ell_{\max}$ **do**
    $c_{\ell} \leftarrow 0$; $\widetilde{c}_{\ell} \leftarrow 0$ ▷ Initial "guesses" for corrections
    $c_{\ell} \leftarrow \text{JACOBI}(A_{\ell} c_{\ell} = b_{\ell}, \omega)$ ▷ Iterate of correction equation stored in c_{ℓ}
    $\widetilde{c}_{\ell} \leftarrow P I c_{\ell}$ ▷ Computation of localised damping for c_{ℓ}
  end for
  $c_{\ell_{\min}} \leftarrow 0$; $\widetilde{c}_{\ell_{\min}} \leftarrow 0$
  $c_{\ell_{\min}} \leftarrow \text{JACOBI}(A_{\ell_{\min}} c_{\ell_{\min}} = b_{\ell_{\min}}, \omega)$ ▷ No auxiliary damping for coarsest level
  $u_{\ell_{\max}} \leftarrow u_{\ell_{\max}} + c_{\ell_{\min}} + \sum_{\ell=\ell_{\min}+1}^{\ell_{\max}} P^{\ell_{\max}}_{\ell} c_{\ell} - P^{\ell_{\max}}_{\ell-1} \widetilde{c}_{\ell-1}$
end function

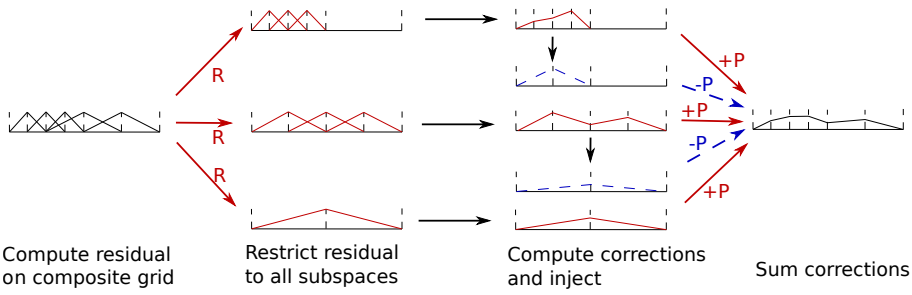

Figure 5.5: Data flow overview of adAFAC-PI. Solid red lines denote traditional subspaces within additive correction equations, dashed blue lines correspond to auxiliary equations that damp the existing correction equations.

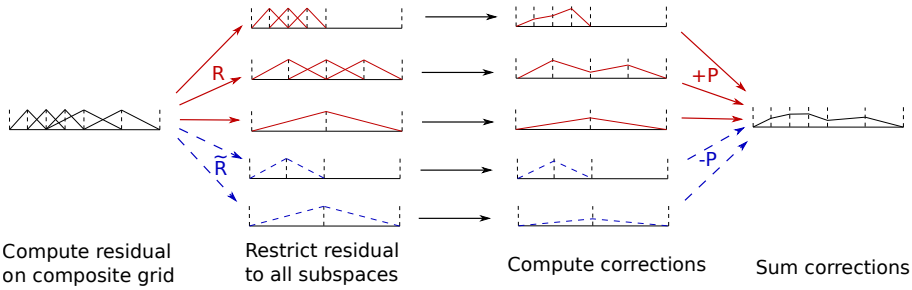

Figure 5.6: Data flow overview of adAFAC-JAC. Solid red lines denote traditional subspaces within additive correction equations, dashed blue lines correspond to auxiliary equations that damp the existing correction equations.

operators $\tilde{R}_\ell^{\ell-1}$ (the latter option is used to construct damping parameters in two different ways):

1. Sole injection where we collapse $\tilde{M}_{\ell-1}^{-1} R_\ell^{\ell-1} A_\ell$ into the identity. The overall damping reduces to $-\omega P_{\ell-1}^\ell I M_\ell^{-1}$. We evaluate the original additive solution update. While we perform this update, we identify updates within c-points, i.e. for vertices spatially coinciding with the next coarser mesh, inject these, immediately prolongate them down again, and damp the overall solution with the result. The damping equation is PI (Algorithm 7). A schematic representation is shown in (Fig. 5.5).
2. A "smoothed" $\tilde{R}_\ell^{\ell-1} \approx \omega R_\ell^{\ell-1} A M_\ell^{-1}$, where we take the transpose of the full operator above and truncate the support, i.e. throw away the small negative

entries by which the stencil support grows. Furthermore, we approximate $\tilde{M}_{\ell-1} = M_{\ell-1}$, i.e. reuse multigrid's correction operator within the damping term. For this choice, memory requirements are slightly increased (we have to track one more "unknown") and two solves on all grid level besides the finest mesh are required (Algorithm 6). The flow of data between grids can be seen in (Fig. 5.6).

3. An alternative "smoothed" approach. We again use a smoothed operator but instead smooth the prolongation operator rather than the restriction. That is, we use $\tilde{P}^{\ell}_{\ell-1} \approx (\omega R^{\ell-1}_{\ell} A M^{-1}_{\ell})^T$ for the auxiliary grid but keep the restriction operator the same ($\tilde{R}^{\ell-1}_{\ell} = R^{\ell-1}_{\ell}$)

Both transfer operators are motivated through empirical observations. Our results study them for jumping coefficients in complicated domains, while previous work demonstrates the suitability of this scheme Helmholtz-type setups [32]. The second option, the intergrid transfer operator with smoother directly applied, is initially only used as a restriction operator—we later explore the possibility of using it instead as a prolongation operator for the auxiliary grid with a standard restriction operator. Though the outcome of both approaches is promising for our tests, we hypothesise that more complicated setups, such as convection-dominated phenomena, require more care in the choice of $\tilde{R}^{\ell-1}_{\ell}$, as $R^{\ell-1}_{\ell}$ has to be chosen more carefully [71].

All approaches can be combined with multigrid with geometric transfer operators where $P^{\ell}_{\ell-1}$—and similarly $R^{\ell-1}_{\ell}$—is d-linear everywhere, or with algebraic approaches where P stems from BoxMG. All approaches inherit Ritz-Galerkin operators on the coarse grids, i.e. $A_{\ell} = R^{\ell}_{\ell+1} A_{\ell+1} P^{\ell+1}_{\ell}$ if they are used in the baseline additive scheme. Otherwise, they exploit redisretisation.

All approaches are able to integrate with HTMG [95]. This allows for a simple implementation of dynamic adaptivity and AMR, by avoiding the explicit handling of the transition between regions that are correction spaces and fine grid spaces. For a FAS-based implementation, all grid levels must be consistent—corrections from all grid levels must be applied to all grid levels. This creates an implicit synchronisation between levels and a two-way flow of information for corrections. Implementation specifics of this follow later. However, the key concept in this melding of techniques is a specific auxiliary correction must be applied to all grid levels—an auxiliary grid correction is applied to all coarse grid levels not just finer grid levels.

Idea 11. *As our solver variants are close to AFAC, we call them adaptively damped AFAC and use the postfix PI or Jac to identify which operators our damping parameters employ. We thus introduce adAFAC-PI and adAFAC-Jac.*

5.4 adAFAC-Jac as a prolongation operator

Our final proposal is to use a smoothed prolongation operator for the auxiliary grid and keep the restriction operator the same as the existing coarse grid equations. This motivation means we now approach the adAFAC-Jac principle from an alternative direction. We had previously attempted to replicate a $V(1,0)$-cycle, instead, we could construct the auxiliary grid along the grounds of a $V(0,1)$-cycle. A simple representation of a $V(0,1)$-cycle is show in Fig. 5.4. This produces the following set of equations:

$$\begin{aligned} u_\ell^{(n+1)} &= \left[u_\ell + \omega_\ell M_\ell^{-1}(b_\ell - A_\ell u_\ell)\right] \\ &\quad + P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_\ell^{\ell-1}(b_\ell - A_\ell u_\ell) \\ &\quad - \omega_\ell M_\ell^{-1} A_\ell P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_\ell^{\ell-1}(b_\ell - A_\ell u_\ell). \end{aligned} \tag{5.4.1}$$

These are the same set of linear operators as in (5.2.1), but they are now re-ordered.

The difference between a two grid level multiplicative and additive scheme is now

$$\begin{aligned} u_{\ell,V(0,1)}^{(n+1)} - u_{\ell,\text{add}}^{(n+1)} &= P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell u_\ell) \\ &\quad -\omega_\ell M_\ell^{-1} A_\ell P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell u_\ell) \\ &\quad -P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell u_\ell^{(n)}) \\ &= -\omega_\ell M_\ell^{-1} A_\ell P_{\ell-1}^{\ell} A_{\ell-1}^{-1} R_{\ell}^{\ell-1}(b_\ell - A_\ell u_\ell). \end{aligned} \tag{5.4.2}$$

We can again approximate the coarse grid solve with a single smoothing step and collapse the additional fine grid smoothing into the intergrid transfer operator. This alternative is closer to the final corrections seen in AFACx and acts as a comparison between adAFAC-x and AFACx.

5.5 Smoothed intergrid transfer construction

Let ϵ in (1.1) be one. We observe that a smoothed operator $M^{-1}A_{\epsilon=1}P$ derived from a generic bilinear interpolation symbol P using a Jacobi smoother $M^{-1} = diag(A)^{-1}$ for three-partitioning corresponds to the stencil

$$\begin{bmatrix} -0.0139 & -0.0417 & -0.0833 & -0.0972 & -0.083 & -0.0417 & -0.0139 \\ -0.0417 & 0 & 0 & 0.0833 & 0 & 0 & -0.0417 \\ -0.0833 & 0 & 0 & 0.167 & 0 & 0 & -0.0833 \\ -0.0972 & 0.0833 & 0.167 & 0.444444444 & 0.167 & 0.0833 & -0.0972 \\ -0.0833 & 0 & 0 & 0.167 & 0 & 0 & -0.0833 \\ -0.0417 & 0 & 0 & 0.0833 & 0 & 0 & -0.0417 \\ -0.0139 & -0.0417 & -0.0833 & -0.0972 & -0.0833 & -0.0417 & -0.0139 \end{bmatrix}.$$

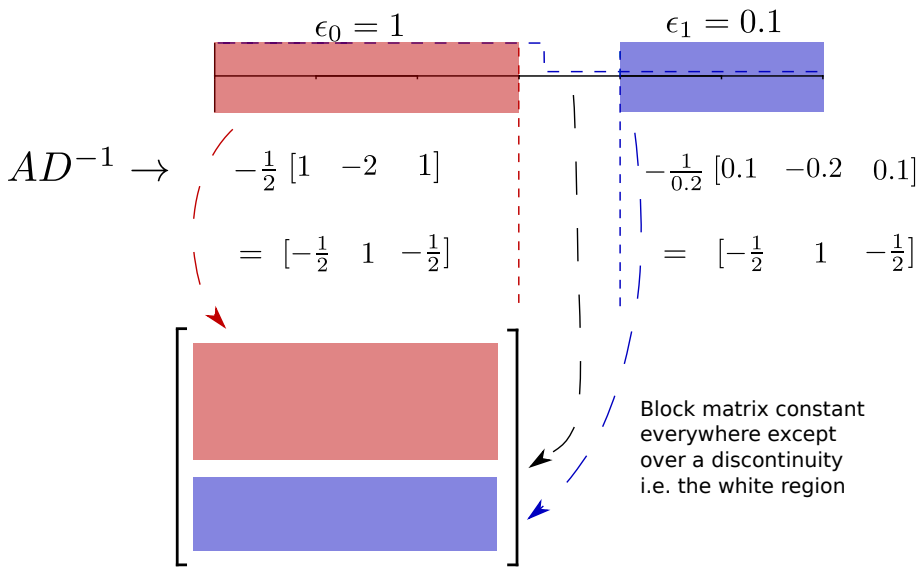

Figure 5.7: Computation of the product AD^{-1} (using stencil notation) for regions of constant ϵ. D is the diagonal of A. The ϵ cancels. We highlight the regions as blocks in the resultant matrix beneath. Both non-white regions hold the same values, a discontinuous ϵ only changes AD^{-1} directly over the discontinuity.

We neglect the damping weight ω in the stencil here. A stencil is a restructured row of the full operator. We also introduce the generic restriction symbol R as the transpose of P, and matrix D as the diagonal of A. The term $A_\ell M_\ell^{-1}$ now enters the auxiliary restriction. Such an expression removes the impact of ϵ on all elements with non-variable ϵ, therefore, for cells with constant ϵ we may treat the expression $A_\ell M_\ell^{-1}$ as constant. Assuming jumps in ϵ are not too significant, we can could use this assumption across the entire domain and only neglect small perturbations in the off-diagonals of the system matrix. In the main, however, we only rely on this assumption for regions of constant ϵ.

This removal of the ϵ influence in regions of constant ϵ when using a Jacobi smoother is illustrated in Fig. 5.7. Constant epsilon over a cell is removed by the inverse diagonal matrix, which is composed of diagonal weights of $\frac{1}{\epsilon}$. This highlights that overcorrections within additive multigrid occur when projecting over a discontinuity. Damping brings greatest benefit over transition regions by cancelling out overcorrections from coarse grids when being projected down to the fine.

5.6 Incorporating other/non-Jacobi smoothers

Similarly to our method of delayed assembly (Section 4.9) at first glance there is no obvious link or effect between the choice of smoother and our ideas; however there may also exist a link on a deeper level. Changing the smoother used effects rates of convergence. A more aggressive smoother, for example Red-Black Gauss-Seidel or ILU, will produce larger corrections to the solution compared to a simple scheme, like Jacobi, and as such, a solver would converge in fewer iterations. This could induce larger oscillations in the solution than Jacobi, if oscillations already exist, that must be damped out. It has been shown that adaptively chosen damping factors are beneficial to more powerful smoothers, this has been shown for an ILU smoother and an anisotropic material parameter [14]. Our damping term should remain effective for such scenarios, but we could not say for certain without further study.

In our current implementation we either exclusively rely on a geometric interpretation of the intergrid transfer operators, or a geometrically inspired choice. We use BoxMG, which is constructed so that corrections projected to the fine grid map directly into the nullspace of the local fine grid operator. In the vein of Brandt, and Compatible Relaxation [110], [111], the coarse relaxation scheme should ensure projected corrections map onto the near-nullspace. From this perspective, the choice of smoother greatly impacts the intergrid transfer operators used.

From an implementational perspective there is also much to consider. The solve on the auxiliary grid does not change its character with different smoothers—in principle it is merely another solve on an existing coarse grid level. Smoothing on the auxiliary grid is therefore implemented the same way as smoothing on the existing coarse grid, and can re-use all existing data structures and functions. However, different smoothers can change the character of intergrid transfer operators drastically. Quite intuitively, a smoother that acts on a larger set of degrees of freedom will increase the number of degrees of freedom that a partially smoothed intergrid transfer operator acts upon. Amongst other issues, this changes the data access pattern, and may

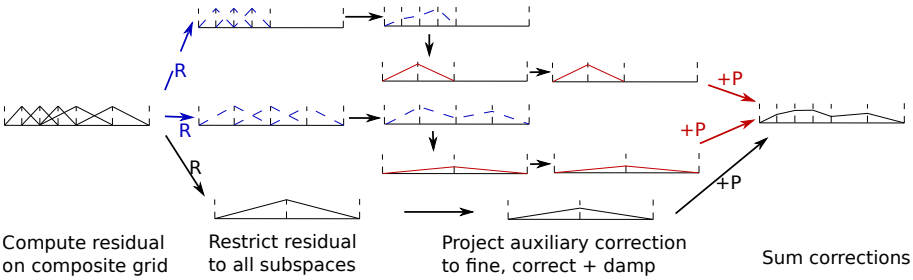

Figure 5.8: Data flow overview of AFACx. Solid red lines denote traditional subspaces within additive correction equations, dashed blue lines correspond to auxiliary equations that damp the existing correction equations.

make certain data structure choices for the grid, such as space-trees, unsuitable. If a smoother can be written as a single matrix application then it is straightforward to construct an adAFAC-x scheme based upon it. Simply precompute an intergrid transfer operator using the product of this matrix representation. adAFAC-x would be a good fit for a polynomial smoother, for example, as it already has a clear matrix representation—it may lead to a complicated data access pattern, however. Other smoothers with a ready matrix representation and fixed data access pattern, such as block smoothers, are also well suited, and produce a partially smoothed intergrid transfer operator that can be hardcoded. adAFAC-x would mesh less well parallel solvers that do not have simple and sparse matrix representations. A Gauss-Seidel smoother would be infeasible to implement in this fashion—precomputing the operator explicitly would render a dense matrix for the intergrid transfer operator. Alternatives such as Red-Black Gauss-Seidel would also be non-trivial. One may have to directly apply the smoother to the residual, then restrict as a two stage process, rather than being able to collapse the smoother application and intergrid transfer operator into a single matrix operator/stencil.

5.7 Comparisons to existing solvers

5.7.1 adAFAC-PI

adAFAC-PI takes inspiration from BPX. The auxiliary solve here is constructed in a similar method to coarse corrections in BPX. BPX builds up its correction solely through intergrid transfer operators while the actual fine grid system matrix does not directly enter the correction equations. Though they do not deliver an explanation as to why the solver converges, the introduction of the PI-scheme in [32] thus refers to this solver as BPX-like. Rather than performing a smoothing step on an auxiliary equation system, adAFAC-PI instead restricts (via injection) the residual and scales it by a (fine grid) diagonal matrix. BPX constructs coarse corrections similarly by scaling coarse equations by a restricted diagonal matrix. This is a very crude, but also very inexpensive, interpretation of a smoothing step. We differ from a pure BPX solver as we build this BPX-like solver over the top of an existing additive multigrid scheme.

Further noteworthy comparisons can be made. adAFAC-PI also shares features with forms of AFAC. Historically, there are two formulations of AFAC: the most common is AFACf, where the fine grid corrections are modified; and the less common is AFACc where the coarse grid equations are modified via the right-hand side. adAFAC-PI shares similarity to AFACc but approaches things the other way round. In AFACc coarse grid contributions are removed when they coincide with fine grid points, this is achieved by setting the right-hand side of the coarse equation to zero. adAFAC-PI instead negates the impact of fine grid points that are topologically in the same space as coarse grid points; this feature is more similar to BPX and other hierarchical basis schemes than AFACc. adAFAC-PI goes beyond just ignoring the impact of theses vertices as their potential impact is removed from neighbouring vertices—it actively negates part of the solution, rather than merely not being applied—this change recovers additional stability. This is empirically shown in Section 7.3.

5.7.2 adAFAC-Jac

Our adAFAC-Jac approach shares ideas with the mult-additive approach [89] where smoothed transfer operators are used in the approximation of a $V(1,1)$-cycle. Mult-additive yields faster convergence as it effectively yields stronger smoothers. We stick to the simple presmoothing approach and solely hijack the additional term to circumvent overshooting, while the asynchronicity of the individual levels is preserved. Implementations of mult-additive, such as the asynchronously smoothed variant by Wolfson-Pou and Chow [90] highlight the increased workload in mult-additive (even compared to traditional multiplicative). Our focus is on keeping the workload per iteration low while still retaining improved convergence. We use only a single smoothed operator and stick to simple smoothers—rather than the more powerful, but more expensive, smoother mult-additive uses.

We also share many similarities to the solvers within the AFAC family (Sect. 3.2.5). We inherit the principle of auxiliary grids damping fine grid corrections which originated with AFACf, and the focus on smoothing steps from AFACx. AFACx incorporates an additional sequential element to improve convergence, the auxiliary smoothing must occur before the coarse grid smoothing, this is illustrated in Fig. 5.8. AFACx does not cause significant degradation in convergence rates due to only performing smoothing [82], [112], validating the use of smoothing steps. We push the asynchronous part of AFAC further, however, and insist that all corrections should be produced independently. Therefore we see similar benefits of improved rate of convergence but with the same potential for parallelism as standard additive multigrid. Using the smoothed intergrid transfer operator as a prolongation operator produces an ordering of operations that is much closer to AFACx. It corresponds to an AFACx implementation with a single pointwise smoothing step on the auxiliary and standard coarse grids. However, we have removed the sequential element between the grid levels.

A direct comparison of specific elements between adAFAC-Jac and other solvers is

shown in Table 5.1.

	Grid Order	Auxiliary Grids	Smoothed R/P	Coarse Solve	Fine Solve (inc. Aux Solve)
Multiplicative	Sequential	✗	✗	$M_{\ell-1}^{-1}$	M_ℓ^{-1}
Additive	Concurrent	✗	✗	$M_{\ell-1}^{-1}$	M_ℓ^{-1}
BPX	Concurrent	✗	✗	$D_{\ell-1}^{-1}$	M_ℓ^{-1}
AFAC	Concurrent	✓	✗	$A_{\ell-1}^{-1}$	$A_\ell^{-1} - PA_{\ell-1}^{-1}R$
AFACx	Concurrent	✓	✗	$M_{\ell-1}^{-1}$	$PM_{\ell-1}^{-1}R - PM_{\ell-1}^{-1}R + M_\ell^{-1}A_\ell PM_{\ell-1}^{-1}R$
Mult-additive	Concurrent	✗	✓	$M_{\ell-1}^{-1} + M_{\ell-1}^{-T} - M_{\ell-1}^{-T}A_{\ell-1}M_{\ell-1}^{-1}$	$M_\ell^{-1} + M_\ell^{-T} - M_\ell^{-T}A_\ell M_\ell^{-1}$
adAFAC-PI	Concurrent	✓	Injected fine correction	$M_{\ell-1}^{-1}$	$M_\ell^{-1} - PIM_\ell^{-1}$
adAFAC-Jac	Concurrent	✓	Auxiliary R only	$M_{\ell-1}^{-1}$	$M_\ell^{-1} - PM_{\ell-1}^{-1}RAM_\ell^{-1}$

Table 5.1: Comparison of key features between existing multilevel solvers and adAFAC-Jac. We use M^{-1} as a generic smoother symbol and D as the diagonal of A.

Chapter 6

Implementation

In Chapter 4 and Chapter 5 we detailed our main algorithmic contributions, but the actual implementation was left open. We now explain our chosen implementation to fill in some of those gaps. Our implementation builds upon the single-touch ideas in Section 3.3.3, so that one iteration of the solve is embedded into a single traversal of a space-tree. This reduces total memory accesses to improve overall performance by keeping arithmetic intensity high. Initially, we cover how our asynchronous assembly process fits into an existing single-touch additive multigrid solve. We introduce additional data structures to achieve this and ensure lazy stencil update process is asynchronous. Subsequently, we cover how damping schemes can be written in a single-touch way. Our baseline additive solver is based on the work of Reps and Weinzierl [32] and we reframe their damping algorithm in terms of auxiliary grids to show it is part of the adAFAC-x class of solvers. We then use this framing and write our adAFAC-Jac implementation in a single-touch fashion.

6.1 Background stencils

In Section 4.3 we outline what individual work units are for our asynchronous assembly process: We construct element-wise stencils based on finite elements. Each element (in a finite element sense) creates a series of tasks/integration quadratures

of increasingly fine subcell size. Each integration quadrature is a discrete work unit. We use Intel's Threading Building Blocks (TBB) [113] for our tasking. Therefore we define a direct mapping between our numerical integration tasks and a task in a TBB sense. We generate TBB tasks while traversing the mesh and add them to a task queue. When a core idles during the overall solve, it pulls an integration task from this queue. Greater numbers of subcells used in a numerical integration increase the accuracy of material data representation. Each mesh element produces 2^d element-wise stencils—one for each neighbouring vertex—a task is thus the numerical integration of all 2^d element-wise stencils. These tasks are independent of each other and all subcells in an element are processed only once to compute all 2^d element-wise stencils. We store stencils persistently within vertices rather than storing as their element-wise contributions within cells. This requires us to reconstruct fine grid stencils whenever a new element-wise update is computed. Element-wise decompositions are not unique, so there is no guarantee that the recomputed decomposition of a previously constructed nodal stencil is the same as when numerically integrating the element-wise stencil. This lack of uniqueness means we cannot perform in-place updates without redundant copies of element-wise stencils. Nodally held stencils update asynchronously once element-wise updates are available—we remove a global synchronisation step. Locally a nodal stencil is only "synchronised" when updated. We reiterate that our smoother is also element-wise and the element-wise stencil decomposition is computed on-the-fly from the persistently held nodal stencils, so smoothing is thus totally independent to the element-wise stencil construction.

6.1.1 Additional data structures

All updated stencils from tasks are stored in a heap. The computational work for an integration task is independent of all other actions within a multigrid solver and new integrations are computed asynchronously. Synchronising updates is therefore not required. A task writes updated stencil entries to the heap (rather than writing to a cell directly) and terminates once done, freeing any memory required for the compu-

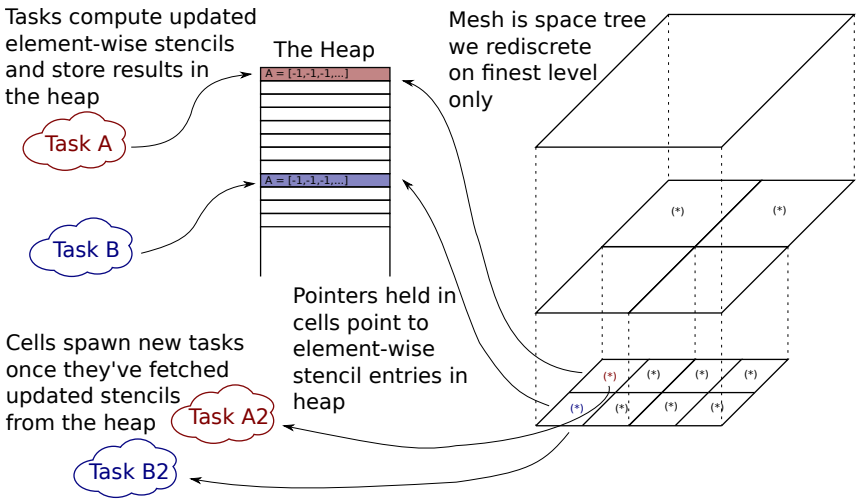

Figure 6.1: Fine grid cells within the space-tree (left) each hold pointers to entries in the heap that stores updated version of the local stencil that result from the background tasks.

tation. This is shown in Fig. 6.1. The use of a heap simplifies the implementation in terms of asynchronicity—a cell holds the index for a location in the heap rather than an element-wise stencil itself. Cells check the heap for updates independently. If updated stencil entries exist, then these updated values are written to a temporary stencil that is held nodally. This temporary stencil is rolled over—i.e. the nodally held stencil is overwritten with this data—after a grid traversal is complete and all cells have been visited. We also introduce two parameters per cell, an atomic flag and an integer n, that are encoded in a tuple: The atomic flag is a signifier of the most recently deployed task having terminated; the n corresponds to the "accuracy" of the element-wise stencil, i.e. a count of how many subcells are used in a numerical integration. In combination, these encode the current state within the sequence of numerical integrations. They can be encoded into the cell stream, similarly to how nodal stencils are stored, and loaded into cache when the cell is accessed.

We traverse the tree and check this tuple once per iteration, seen in Fig. 6.2. There are three possible branches based on the state of n:

- $n = \perp$: This is the first time the cell has been accessed in a solve, i.e. there is no previously existing element-wise stencil. We therefore compute an initial

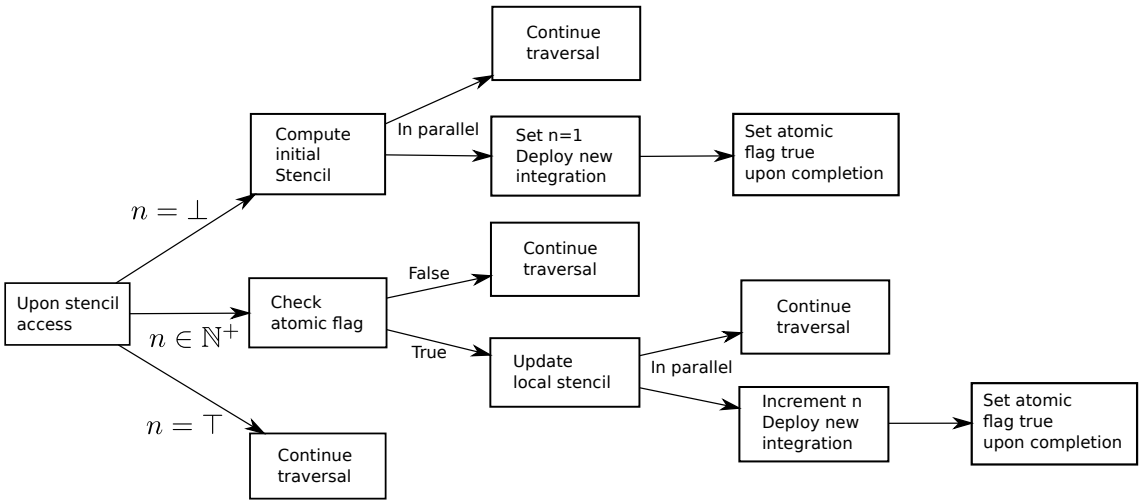

Figure 6.2: Illustrative diagram of how we perform the lazy integration. All cells carry a n that holds the number of samples per dimension of the quadrature.

stencil—this is a serial operation and is computed before the grid traversal continues. For our purposes, this initial stencil is the already known Poisson stencil weighted by a single material parameter sampling point. This is used in initial smoothing steps, and is thus inserted into the hashmap immediately. We now branch and can perform the following two steps in parallel:

1. The traversal of mesh elements in the tree continues, using the stencil in the hashmap.
2. An additional numerical integration task, with 2^d subcell sampling points, is deployed. This is not processed immediately, but sent to a task queue and handled by a core that may otherwise be idle. We initially set the atomic flag to false to signify there is no current update, and the task then sets this to true upon completion. n is also set to 2 to represent the increased accuracy of the integration.

- $n \in N$: The stencil is actively being improved but is not sufficiently accurate thus far. As such, there may possibly be an updated stencil available to be rolled over into the nodal stencil. We check the atomic flag, if this returns false we simply carry on with mesh traversal. However, if this returns true then we know the previously deployed task has terminated, and similarly to

the previous case, we branch again. If the atomic flag is true, we perform the following two steps in parallel:

1. We continue mesh traversal, using the currently held stencil values in the heap. However, vertex-wise stencils may be required to be reconstructed if neighbouring stencils have not finished their sequence of numerical integrations.
2. We re-accumulate the nodal stencil with the update element-wise stencil—after re-accumulation the nodal stencil is rolled over. During this process, we compute a matrix norm to check if the stencil is of sufficient accuracy. For previous element-wise stencil $A^{(old)}$, updated element-wise stencil $A^{(new)}$, and the maximum element-wise matrix norm $\|.\|_{\max}$, we set

$$n \leftarrow \begin{cases} \top & \text{if } \frac{\|A^{(\text{new})} - A^{(\text{old})}\|_{\max}}{\|A^{(\text{old})}\|_{\max}} < C \\ n+1 & \text{if } \frac{\|A^{(\text{new})} - A^{(\text{old})}\|_{\max}}{\|A^{(\text{old})}\|_{\max}} \geq C \end{cases}$$

for a fixed constant C. If $n \leftarrow n+1$, a new numerical integration task is deployed—the atomic flag is again set to false. Otherwise, if $n \leftarrow \top$, the stencil is deemed accurate enough so no further integrations are required.

- $n = \top$: The nodal stencils are accurate enough, therefore we can immediately continue the mesh traversal using the values on the heap.

For a static regular grid, once an element-wise sequence has terminated, processing for that element is finished forever. However, within an AMR context, this no longer holds. After a refinement, a cell may switch from holding a fine grid element to a coarse grid element. Therefore, the semantics of the local equation have changed. Instead of the currently held fine grid equation being accurate, a new coarse grid equation (most likely set by Ritz-Galerkin) is computed and stored. This involves a recomputation step which we cover in the next section.

6.1.2 Coarse grid operators

We set coarse grid equations using a Ritz-Galerkin definition, $A_\ell = RA_{\ell+1}P$. As we use a locally regular Cartesian mesh, this is equivalent to rediscretisation for geometric R and P. Therefore, once the fine grid operators are rolled over at the end of a cycle, the coarse grid operators are now invalidated, as they may have become poor approximations. Any change to an individual element-wise operator on the fine grid invalidates certain regions of the currently held coarse grid—the precise definition of the coarse grids has changed. We assume any changes to be minimal and that we are therefore able to use the old, now outdated, operators. However, as we can not guarantee this, we still must recompute. There are two different operators that must "ripple" up to the coarse grid levels—the coarse grid operators, and, if we use a BoxMG environment, the intergrid transfer operators.

Due to our single-touch methodology, vertex-wise operators can only be rolled over after the multigrid iteration they were set—all neighbouring cells must have been processed for a vertex to be updated. Therefore, in order for operators on the next coarsest grid to incorporate this information, they can only be computed on the subsequent traversal or later. We apply this argument recursively. Thus, this gives a hierarchical set of constraints on coarse grid operators—coarse grid operator information can only propagate one level up the coarse hierarchy each grid traversal. Moreover, in order for updated fine grid equations to enter the updated BoxMG intergrid transfer operators, then the full patch of fine grid vertex stencils must already be updated/rolled over. That is, all fine grid cells that are children of the same coarse grid cell (and their neighbours) must be rolled over for this information to be incorporated into a recomputed BoxMG transfer operator that acts on that coarse grid cell. This is also impossible to guarantee in the same iteration a fine grid stencil is updated, again due to our single-touch policy. Both requirements can therefore be phrased as a partial ordering between grid cell processes. A patch of grid cells must be updated before intergrid transfers can be computed on the next

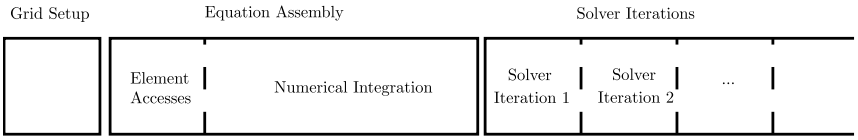

Figure 6.3: Conventional sequential matrix equation assembly. The mesh is assembled and then exact numerical integration of equations is performed before the solver iterations begin.

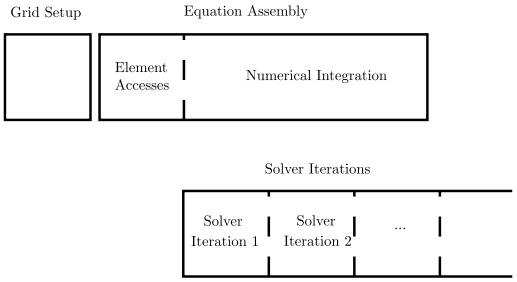

Figure 6.4: Our delayed matrix equation assembly. The mesh is assembled and then exact numerical integration of equations is performed in parallel with the early the solver iterations.

coarser level. Updated intergrid transfer operators are required before updated coarse grid stencils can be computed. All coarse grid transfer operators must therefore be set up prior to the computation of a coarse grid stencil for the representation to be consistent. We cannot ensure this across all levels and all grid cells in a DFS traversal, but we can ensure this locally—fine grid cells are updated prior to updating their direct parents. This dictates our partial ordering.

6.1.3 Performance model

We now briefly outline a performance model to indicate expected gains due to our delayed assembly. The runtime of any multigrid algorithm can be broken down into three stages:

1. Grid setup
2. Equation assembly
3. Solver iterations

These are shown diagrammatically in (Fig. 6.3). With our asynchronous assembly, we

increase the potential for parallelism by reducing the time lag/algorithmic latency due to equation assembly as much as possible. The time cost due to equation assembly is hidden behind the computation of the solver iterations. This means, we expect to see the runtime reduced by a consistent proportion by switching to our assembly methodology, as we are effectively eliminating this phase's impact on the overall runtime.

Each phase in the multigrid iteration has an expected cost which we give relative to the computational cost of a grid traversal. The grid setup is a fixed cost and one that cannot be avoided (we neglect AMR setups for this approximation). As we initially construct each element/cell within the grid, we estimate that the cost of grid setup is roughly proportional to the cost of a grid traversal of the full grid. The solver iterations themselves again are a cost that we do not eliminate with our methodology, and again we approximate that each iteration of the solver has a cost proportional to the total number of elements, i.e. to a grid traversal. Therefore the total cost of the solver iterations corresponds to the number of iterations required for convergence multiplied by the cost of a single grid traversal. The cost our approach explicitly avoids is the equation assembly cost, which can now be performed in parallel with other steps, shown in (Fig. 6.4). and can be further split into two parts: The cost of accessing each element and the actual cost of the integration. Again the total cost of accessing each element once is proportional to the cost of a grid traversal, nut the cost of the integrations is both implementation dependent and potentially unbounded. We therefore refer to the setup cost of other implementations as a guideline. We refer to work by Lin et. al. [92], where the setup cost in terms of time for the smoothers/equations is shown to be within the same order of magnitude as that of the solve itself—for small problem sizes the smoother setup cost dominates the time-to-solution, while for larger problems the smoother setup cost is roughly half that of the solve cost in terms of time. In line with reported runtimes for different phases for other solvers, we could reasonably expect to see an overall reduction in runtime of approximately a third.

6.2 Additive damping

Without careful planning of intergrid transfers and the introduction of helper variables, an iteration of our additive damping scheme would not readily map onto a single traversal of a space-tree. Computing coarse grid corrections and auxiliary grid corrections might require additional data movement and traversals through the grid hierarchy. We therefore use a single-touch methodology to avoid this. A single-touch implementation of an additively damped scheme is showcased by Reps and Weinzierl [32]. This algorithm was there referred to as "BPX"—we have now identified this as a specific form of an adAFAC-x scheme. It is nothing other than our adAFAC-PI algorithm, as the damping parameters can be shown to be be corrections from auxiliary grids. We reiterate their implementation here, but clearly reframe it as an adAFAC-x scheme. Our adAFAC-Jac implementation builds on their method of damping parameter construction and exploitation of space-trees.

6.2.1 Single-touch

adAFAC-PI adAFAC-PI has an implicit auxiliary correction space, but the auxiliary correction space is merely injected updates from the fine grid. We do not perform an additional solve. The corrections from the fine grid are re-used on the auxiliary grid. For c-points (fine grid points that are in the same geometric/topological position as a coarse grid point), the fine grid updates are not applied. This update is computed and fed into the "auxiliary grid", which can then be projected back onto the original grid to be used as a damping parameter for the original corrections. The damped corrections are then (recursively) projected onto the fine grid/composite grid. In a FAC context, we would be done. However, for a HTMG implementation, updates must be synchronised between all levels.

Reps and Weinzierl wrote the single-touch HTMG Algorithm 8 (originally printed in [32] we reprint it here for clarity). Modifying the undamped additive HTMG method to be single-touch required the introduction of a helper variable. Potential

Algorithm 8 Outline of single-touch adAFAC-PI. sc is the summed coarse grid correction contributions. sf is the summed fine grid correction contributions. A tilde identifies variables related to the auxiliary adAFAC grid. $S(u_\ell, b_\ell)$ is the smoother applied to u_ℓ A point v is a *cPoint* if a coarse grid point also exists at the same position in space. We invoke the cycle passing in the coarsest grid ℓ_{min}.

function ADAFAC-PI(ℓ)
  $sc_\ell \leftarrow sc_\ell + P^\ell_{\ell-1} sc_{\ell-1}$ ▷ Prolong contributions from coarse grid
  if *not* $cPoint(v)$ **then**
    $sc_\ell \leftarrow sc_\ell - P^\ell_{\ell-1} \widetilde{sc}_{\ell-1}$ ▷ Damp with correction from auxiliary grid
  end if
  $u_\ell \leftarrow u_\ell + sc_\ell + sf_\ell$ ▷ Anticipate fine grid smoothing
  $\hat{u}_\ell \leftarrow u_\ell - P^\ell_{\ell-1} u_{\ell-1}$ ▷ Determine hierarchical solution
  if $\ell < \ell_{max}$ **then**
    ADAFAC-PI($l+1$)
  end if
  $r_\ell \leftarrow b_\ell - A_\ell u_\ell$ ▷ Compute residual
  $\hat{r}_\ell \leftarrow b_\ell - A_\ell \hat{u}_\ell$ ▷ Compute hierarchical residual
  if $cPoint(v)$ **then**
    $sc_\ell(v) \leftarrow 0$ ▷ Cancel out update
  else
    $sc_\ell \leftarrow \omega S(u_\ell, b_\ell)$ ▷ Perform coarse smoothing
  end if
  if $\ell > \ell_{min}$ **then**
    $\widetilde{sc}_{\ell-1} \leftarrow I\left(\omega S(u_\ell, b_\ell)\right)$ ▷ Inject anticipated update onto auxiliary grid
    $b_{\ell-1} \leftarrow R^{\ell-1}_\ell \hat{r}_\ell$
    $sf_{\ell-1} \leftarrow I(sf_\ell + sc_\ell)$
  end if
end function

updates need to be bookmarked between iterations before they are recursively applied to all finer grid levels— they are not just applied to the level they were computed on. Without this local bookmarking an explicit synchronisation step would be required so that all solution representations are kept consistent. Coarse grid updates are projected to the fine, and applied to both the fine solutions and the bookmarked fine grid updates. This bookmarking means that when fine grid updates are applied to solutions on increasingly finer grids, the coarse updates are also propagated. Fine grid updates are applied to the coarse grid by the fine grid solution value being injected to the coarse when the residual is restricted.

This motivates the introduction of an additional—second—helper variable that solely memorises the fine grid updates injected to the coarse grid and are not applied on

the fine grid. The additional variable simultaneously serves as the correction for the auxiliary grid space and another component of the synchronisation of solution representations across grid levels. It holds potential/unapplied fine grid updates after the updates are injected onto the coarse grid. These updates are then recursively projected onto all finer grids, in the same fashion as the coarse corrections. Within our single-touch framework (see Section 3.3.3), we restrict the fine grid residual to the coarse grid right-hand side at the end of a grid traversal when writing data to main memory. With the fine grid residual at hand we can compute a partial correction for a cell/patch. We use pointwise smoothers, specifically Jacobi smoothers, so once we have computed a residual, we can immediately compute a partial correction and inject to the auxiliary coarse grid. Although we are injecting this update—rather than performing a full restriction—we still perform this restriction at the end of a grid traversal when we are already restricting the residual. When projecting auxiliary corrections in the downward grid sweep of the next traversal we recursively project the updates via this helper variable. Upon first access of a fine grid vertex, we compute the projected impact of both standard and auxiliary corrections from coarse grid vertices. We apply the auxiliary correction to the fine grid solution as a damping parameter (except at c-points). The impact of the auxiliary correction is also applied to the helper variable on the fine grid. This allows the damping parameter to recursively effect all grid levels, keeping the solution consistent across grid levels.

adAFAC-Jac The considerations to rewrite adAFAC-Jac as single-touch have led us to write Algorithm 9. adAFAC-Jac does not just re-use existing corrections—auxiliary corrections must be calculated. Unlike the simple data movement of adAFAC-PI, we perform meaningful auxiliary coarse grid computations for adAFAC-Jac and thus explicitly introduce the additional correction space. Even though we are working with FAS derived HTMG, we only compute corrections for the auxiliary coarse space, we do not require an additional full solution representation. An extra mat-vec is required for the auxiliary smoothing. This is in addition to the

Algorithm 9 Outline of single-touch adAFAC-Jac. sc is the summed coarse grid correction contributions. sf is the summed fine grid correction contributions. A tilde identifies variables related to the auxiliary adAFAC grid. $S(u_\ell, b_\ell)$ is the smoother applied to u_ℓ We invoke the cycle passing in the coarsest grid ℓ_{min}.

function adAFAC-Jac(ℓ)
  $sc_\ell \leftarrow sc_\ell + P^\ell_{\ell-1} sc_{\ell-1}$ ▷ Prolong contributions from coarse grid
  $sc_\ell \leftarrow sc_\ell - P^\ell_{\ell-1} \widetilde{sc}_{\ell-1}$ ▷ Damp with correction from auxiliary grid
  $u_\ell \leftarrow u_\ell + sc_\ell + sf_\ell$ ▷ Anticipate fine grid smoothing
  $\hat{u}_\ell \leftarrow u_\ell - P^\ell_{\ell-1} u_{\ell-1}$ ▷ Determine hierarchical solution
  $\widetilde{u}_\ell \leftarrow 0$ ▷ Reset auxiliary solution
  if $\ell < \ell_{max}$ **then**
    adAFAC-Jac($l+1$)
  end if
  $r_\ell \leftarrow b_\ell - A_\ell u_\ell$ ▷ Compute residual
  $\hat{r}_\ell \leftarrow b_\ell - A_\ell \hat{u}_\ell$ ▷ Compute hierarchical residual
  $sc_\ell \leftarrow \omega S(u_\ell, b_\ell)$ ▷ Perform coarse smoothing
  $\widetilde{sc}_\ell \leftarrow \tilde{\omega} S(\widetilde{u}_\ell, \widetilde{b}_\ell)$ ▷ Perform auxiliary smoothing
  if $\ell > \ell_{min}$ **then**
    $b_{\ell-1} \leftarrow R^{\ell-1}_\ell \hat{r}_\ell$ ▷ Restrict hierarchical residual to coarse grid
    $\widetilde{b}_{\ell-1} \leftarrow \widetilde{R}^{\ell-1}_\ell r_\ell$ ▷ Restrict residual to auxiliary grid
    $sf_{\ell-1} \leftarrow I(sf_\ell + sc_\ell + \widetilde{sc}_\ell)$
  end if
end function

two existing mat-vecs for smoothing the original solution and HTMG's hierarchical residual. We can use the adAFAC-PI specific helper variable for this value; however, this additional variable is now incorporated in computations rather than exclusively data movement. The element-wise residual required to smooth this variable is performed in the same computational step as the two existing residual mat-vecs. A second residual restriction is also required for the additional smoothing. The auxiliary coarse grid right-hand side is set by a smoothed restriction of the residual, as opposed to restricting the hierarchical residual in the base HTMG implementation. This additional restriction is handled in the grid backtracking step, in the same fashion as the original restriction in the baseline single-touch algorithm. To ensure all grid levels maintain a consistent solution representation, the auxiliary corrections must damp all grid levels, not just the finest. Auxiliary corrections therefore also damp the solution representation on their level. When fine grid solutions are injected to coarse grids, the auxiliary corrections are effectively also being applied there by

proxy.

Performance expectations Like any additive multigrid algorithm we expect the cost of an iteration to scale proportionally to the number of elements on the fine grid. Multigrid introduces coarse grid vertices. However, due our geometric coarsening procedure, which coarsens in each dimension by a factor of three, we do not introduce significantly more vertices for a two dimensional grid. The cost is still $O(n)$. E.g. for a fine grid with five million fine grid degrees of freedom (4,778,596) we would introduce around half a million coarse grid vertices across all levels (595,692), when constructing coarse grids using three-partitioning. Approximately a 10% increase in vertices to be processed. Due to the additive nature of our algorithm, these can be handled wholly in parallel. Furthermore, due to our reuse of data structures for our adaFAC-x implementation, we do not anticipate a significant increase in cost of an iteration of our adaFAC implementation relative to a baseline additive multigrid iteration. We might reasonably expect the runtime of a single iteration to have an increase of a few percent. Therefore, we believe the cost of a single grid traversal to be a reasonably good indicator of the cost of one iteration of our algorithm.

6.2.2 Intergrid transfer operators

For adAFAC-Jac we must further augment our implementation—adAFAC-Jac requires smoothed intergrid transfer operators. We can re-use the existing coarse grid space for the auxiliary grid space, i.e. the next coarsest level in a space-tree. It is in those coarse grid vertices that we store the additional auxiliary variable that we use to compute auxiliary corrections. An ideal intergrid transfer would result from a tentative transfer operator that has been smoothed an infinite number of times. This is impractical due to the simple fact that smoothing an operator an infinite number of times is impossible. Applying a smoother to the transfer operator a large number of times on the other hand, is a possible but impractically expensive proposition that would render a transfer operator dense. As well as being costly to compute, a

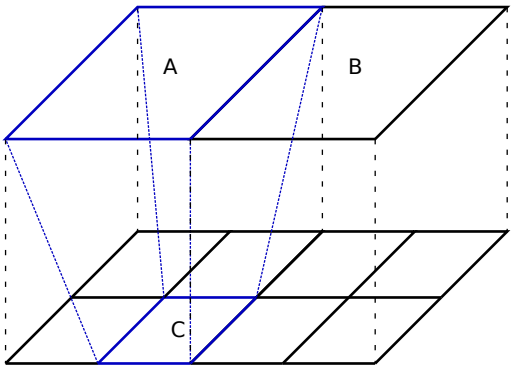

Figure 6.5: In our implementation, we truncate transfer stencils so that vertices adjacent to cell C only restrict to vertices adjacent to parent cell A. There is no transfer to cell B.

dense intergrid transfer is likely to cause a coarse grid solve to become exceedingly expensive. The coarse grid solve with dense intergrid transfers effectively becomes an exact inversion of the matrix—this undermines the multigrid principle of coarse solves being cheap. To reduce the cost of the grid solve we want to use simpler and sparser intergrid transfers. We only apply the smoother once to the intergrid transfer operator and truncate the support to further reduce the cost.

Reducing the size of the support has many benefits—this is a pattern seen in many codes [114], [115]—a minor benefit being the reduction in the operation count of a grid transfer and a major benefit being that the transfer operators are able to retain data locality. A wider support can involve non-local data movement, which adds to memory access times. If we truncate the support we keep data movement largely the same and can plug in to existing data transfers. This focus on data locality can also be seen in our choice of fine grid stencils and smoothers—fine stencils are all constructed using basis functions with local support and we use Jacobi smoothers. When we apply a smoother to an already computed residual, we incorporate no additional information, that is data movement, from other vertices.

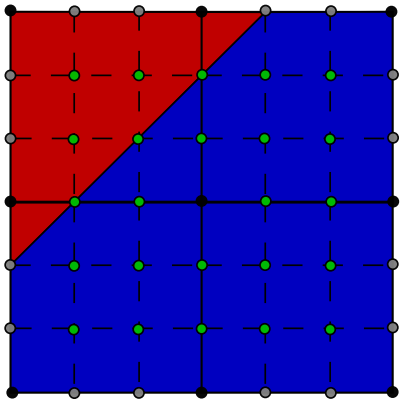

Figure 6.6: Material parameter used for truncated RAM^{-1} computation. The red region (top left) holds material parameter of 0.01 and blue (lower right) holds 1. Black nodes are coarse grid vertices and green nodes are interior points that we retain.

When traversing a space-tree, we have ready access to a geometrically local subset of coarse grid vertices from the perspective of a fine grid cell (see Fig. 6.5). This corresponds to the coarse grid vertices that are adjacent to coarse grid parent cell. Smoothing restriction operator widens the support of a stencil—a restriction operator applied to a patch of 3^d point fine grid stencils will increase the input vertices to the smoothed restriction operation by two in each dimension. This motivates us to not use the true operator. Instead we use a truncated intergrid transfer stencil and throw away outer stencil elements. Note, that in our current implementation we simply discard these elements—formally we should lump these elements and add them to the diagonal. We have observed no negative effects due to our choice here, but this should require further study. Data movement now falls neatly into a space-tree—it is limited to transfers between fine grid cells and their direct parents. When analysing the smoothed restriction stencil for constant epsilon (Section 5.5) we noted that the boundary vertices contribute very small values—we therefore assume we can simply neglect them.

For a region of constant material parameter the stencil for an intergrid transfer operator that restricts to/projects from a single coarse grid vertex, neglecting the

damping weight ω is:

$$\begin{bmatrix} -0.0139 & -0.0417 & -0.0833 & -0.0972 & -0.083 & -0.0417 & -0.0139 \\ -0.0417 & 0 & 0 & 0.0833 & 0 & 0 & -0.0417 \\ -0.0833 & 0 & 0 & 0.167 & 0 & 0 & -0.0833 \\ -0.0972 & 0.0833 & 0.167 & 0.444444444 & 0.167 & 0.0833 & -0.0972 \\ -0.0833 & 0 & 0 & 0.167 & 0 & 0 & -0.0833 \\ -0.0417 & 0 & 0 & 0.0833 & 0 & 0 & -0.0417 \\ -0.0139 & -0.0417 & -0.0833 & -0.0972 & -0.0833 & -0.0417 & -0.0139 \end{bmatrix}.$$

This was already printed in Section 5.5, but we repeat it here for clarity. This smoothed restriction operator can be hard-coded when ϵ is constant; however this does not hold once ϵ varies as sharp jumps in ϵ induce changes in derived operators. Due to our use of BoxMG, we already explicitly store intergrid transfer operators—computing them on-the-fly becomes too costly. We compute and explicitly store smoothed restriction operators across discontinuities (at the very least). The operators are stored as nodal stencils: A restriction operator is stored in the coarse grid vertex it restricts to, and a prolongation operator in the coarse grid vertex it projects from. We need only store one operator per-vertex however, as we use transposed prolongation operator for restriction and prolongation.

We show an example transfer operator, for a chosen material parameter setup (shown in Fig. 6.6). This stencil is computed once and stored between iterations.

$$\begin{bmatrix} -0.0139 & -0.0160 & -0.0168 & -0.0393 & -0.0794 & -0.0456 & -0.0139 \\ -0.0160 & 0.106 & 0.109 & 0.0733 & -0.0322 & -0.00392 & -0.0417 \\ -0.0168 & 0.109 & -0.0279 & 0.0262 & -0.0565 & -0.00392 & -0.0833 \\ -0.0393 & 0.0733 & 0.0262 & 0.366 & 0.155 & 0.0833 & -0.0972 \\ -0.0794 & -0.0322 & -0.0565 & 0.155 & 0 & 0 & -0.0833 \\ -0.0456 & -0.00392 & -0.00392 & 0.0833 & 0 & 0 & -0.0417 \\ -0.0139 & -0.0417 & -0.0833 & -0.0972 & -0.0833 & -0.0417 & -0.0139 \end{bmatrix}.$$

We only store the entries that restrict values from element interiors (shown in green). The outer elements are thrown away. Note, only entries near the discontinuity differ from the stencil shown for constant ϵ, entries in regions of constant ϵ remain the same. Specifically, if the stencil held in a vertex overlaps with a discontinuity, then elements

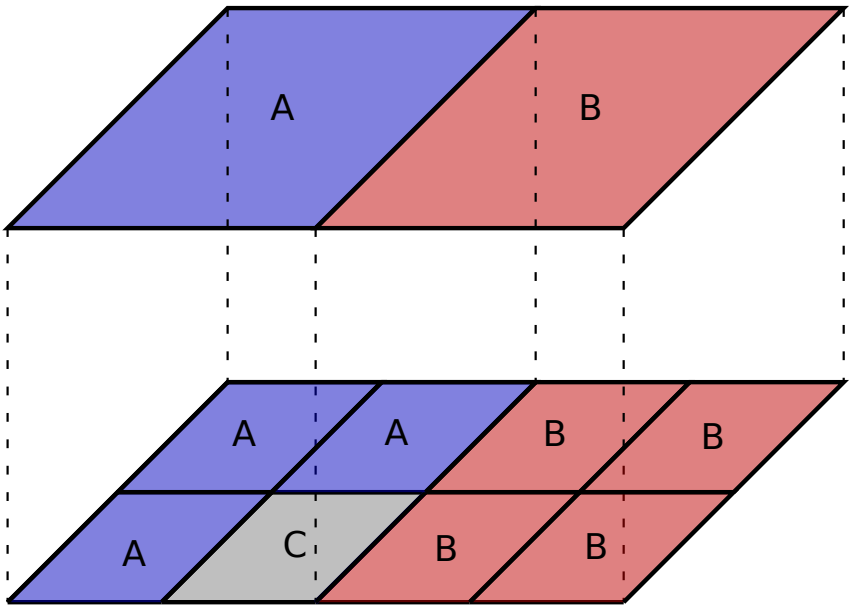

Figure 6.7: Decomposition of a space-tree into subdomains. Cells are assigned to rank A, B or C.

in the intergrid transfer operator that correspond to the vertices that influence the stencil are altered.

6.2.3 Extending to distributed memory implementation

To effectively run on modern large scale machines an implementation must be able to run on distributed memory systems. It is a rather straightforward exercise to split any algorithm that acts on a space-tree into a distributed memory implementation that acts on decomposed domains. A space-tree can readily be topologically decomposed into different subdomains that can then be assigned to different processors, or more specifically MPI ranks for our target implementation. An MPI rank is assigned a coarse cell and a selection of its child cells. MPI ranks do not hold disjoint regions of cells. A cell is only assigned to an MPI rank if it is either the coarsest cell in that branch of a tree, or its parent is also assigned to that rank. An example decomposition can be seen in (Fig. 6.7). As our implementation already handles cells in an elementwise fashion while traversing the grid, the distributed memory implementation can traverse the section of the tree it holds in much the same way. The only wrinkle is how to explicitly handle degrees of freedom on the boundary of

each domain.

Degrees of freedom that lie directly on the boundary are held redundantly between ranks. For example, in (Fig. 6.7), degrees of freedom on the edges of the cell assigned to rank C would be held on rank C with select degrees of freedom also existing redundantly on rank A and B. Each rank acts on its own local copy and ranks exchange updates between iterations. As we use an elementwise traversal, we are not required to introduce ghost elements along the boundary, and instead, during the traversal for a specific rank, we accumulate partial sums for relevant values—i.e. the three residuals we require—based on the processing of local elements. When updates are exchanged between iterations these partial sums are accumulated and become full sums. Partial residuals can be restricted to locally held coarse cells per rank and coarse cells are able to act upon these (this is possible due to the additive nature of restriction). For certain scenarios, such as rank C in (Fig. 6.7), we must introduce redundant coarse degrees of freedom that can be restricted to.

6.3 Wrap up and limitations of current concurrency

In an ideal case, the increase in concurrency due to tasking with the asynchronous assembly is $O(n)$. Each stencil construction operation is its own task and all tasks are totally independent, but for most setups there is likely to be rapid drop off in the number of tasks within the system. Fine grid stencils for most topological regions will require only a small number of integration iterations before they are deemed converged. The majority of the additional concurrency is only seen earlier in the solve. Later concurrency gains are only for regions that require a large number of stencil iterations before they terminate. A large improvement can be seen with dynamically adaptive meshes, once the number of fine grid elements increases throughout the solve—therefore the number of cells/elements requiring accurate integration, and

the number of assembly tasks, also increases. Additional concurrency is thus seen throughout the solve, or at least while refinements are ongoing.

For our implementation, we embed the vertical rippling directly into the mesh traversal. As we traverse the mesh, we recompute the coarse grid operators when we write updates for cells that hold the operators back to main memory. They are recomputed each traversal. This does not take into account another possible avenue for concurrency: An actor system could be employed so that coarse grid recomputations are also a series of, potentially concurrent, tasks. Coarse grid stencils should only be recomputed when there is fine grid detail they do not already incorporate and these recomputations could be deployed to the background as a series of tasks.

Formally, when truncating our partially smoothed intergrid transfer operator, we should lump the discarded stencil elements onto the diagonal. We do not currently do this. Although we have observed no deterioration in convergence rates due to our choice, this does not align with theory from smoothed aggregation multigrid, so warrants further investigation. Our damping parameters are constructed as a correction to a correction—modifications to the real solution are thus indirect and two steps removed. This may be why damping parameters being "off" makes no discernible difference to the solution. Furthermore, we also use damp correction equations with ω—and further damp auxiliary corrections with ω^2 through the partially smoothed intergrid transfer—this aggressive damping might be another reason we do not need to lump stencil elements.

Chapter 7

Results

In previous chapters we covered our ideas from a theoretical perspective (Chapter 4 and Chapter 5) and our target implementation (Chapter 6). Now we test those ideas and provide the results of those tests. We have developed two main contributions—our additive damping parameter adAFAC-x solvers and our asynchronous assembly method—which we use to group results into two high level groups. Within each group, we analyse that contribution first in terms of consistency, then stability and finally in terms of performance. We present the adAFAC-x results first. This order is the inverse of the earlier sections as we specifically investigate the impact adAFAC-x has on the different assembly methods

The following chapter is modified from text that was previously published in [1]–[3]. The introduction and problem setup incorporates elements of all three papers. The three Sections 7.2-7.4, the analysis of our damping parameter, is an expanded version of the results section from "Stabilised Asynchronous Fast Adaptive Composite Multigrid using Additive Damping". The subsequent three Sections (7.5-7.7) are an intermixed combination and expansion of the results section from "Lazy Stencil Integration in Multigrid Algorithms" and "Delayed approximate matrix assembly in multigrid with dynamic precisions". For all six sections, the majority of the data in graphs is from slightly modified setups than those printed in the previous papers—regularity conditions have been changed. The code used here has been made available at https://bitbucket.org/CDMurray/adafacx/src/master/.

	Boundary condition	Right-hand side
BC1	$u\|_{\partial\Omega} = sin(\pi x_0)$ for $x_1 = 0$ $u\|_{\partial\Omega} = 0$ otherwise	$f = 0$
BC2	$u\|_{\partial\Omega} = 0$	$f = -2\pi^2 sin(\pi x_0) sin(\pi x_1)$
	ϵ values	ϵ boundaries
E1	1	N/A
E2	$\epsilon \in \{1, 10^{-k}\}$, $k \in \mathbb{N}$	$x_0 = 0.5$
E3	$\epsilon \in \{1, 10^{-k}\}$, $k \in \mathbb{N}$	$x_1 = 5x_0 - 2.5$ and $x_1 = 0.2x_0 + 0.5$

Table 7.1: Summary of the features we change in the equations. We use two different pairings of boundary conditions and right-hand sides (BC1/BC2) and compare three different sample ϵ distributions with k fixed per run.

7.1 Experimental setup

7.1.1 Test hardware

All experiments are run on an Intel Xeon E5-2650V4 (Broadwell) with 12 cores per socket clocked at 2.4 GHz. As we have two sockets per node, a total of 24 cores per node is available. These cores share 64 GB TruDDR4 memory. Shared memory parallelisation is achieved through Intel's Threading Building Blocks (TBB), We rely on a TBB wrapper [116], [117] with a custom priority layer such that we have very fine-granular control over which tasks are run when.

7.1.2 Scenarios and test equations

All our studies solve the variable coefficient Poisson equation (1.1), specifically

$$-\nabla(\epsilon \cdot \nabla)u = f$$

is solved on the unit square for differing values of ϵ. We introduce three different material parameter configurations: constant ϵ and two discontinuous setups. For the first, and simplest, case (E1), we fix $\epsilon = 1$ everywhere—this is merely the

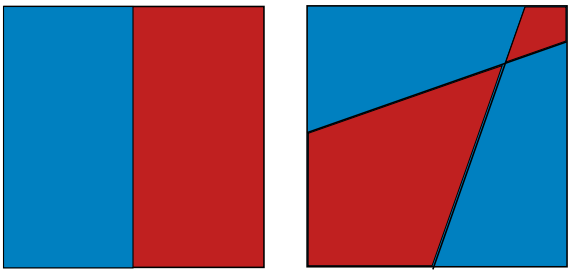

Figure 7.1: The two non-constant ϵ distributions studied throughout the tests. Left: (E2). Right: (E3). The blue area holds $\epsilon = 1$, while the remaining domain holds $\epsilon = 10^{-k},\ k \in \{1, 2, \ldots, 5\}$.

Poisson equation. In terms of the two discontinuous setups, we split the domain into disjoint regions, which hold either $\epsilon = 1$ or $\epsilon = 10^{-k}$. Per run, the respective $k \in \{1, 2, \ldots, 5\}$ is fixed. The first of the two discontinuous setups (E2), introduces the discontinuity along the line $x_0 = 0.5$. The second discontinuous setup (E3) separates the subdomains via the lines $x_1 = 5x_0 - 2.5$ and $x_1 = 0.2x_0 + 0.5$, and each adjacent subdomain holds a different ϵ in a checkerboard fashion. No parameter split is axis-aligned. These are simplistic yet already challenging for multigrid. We solve for two different pairings of boundary conditions and f values. In the first pairing (BC1), we set $f = 0$ over the whole domain, and set $u|_{\partial\Omega} = sin(\pi x_0)$ for $x_1 = 0$ or $u|_{\partial\Omega} = 0$ otherwise. For the other pair (BC2), we instead use homogeneous boundary conditions, that is $u|_{\partial\Omega} = 0$, and $f = -2\pi^2 sin(\pi x_0) sin(\pi x_1)$ over the entire domain.

Our initial consideration is consistency studies for the adAFAC-x solver suite. We test adAFAC-PI and adAFAC-Jac—both using partially smoothed restriction and a variant that uses partially smoothed prolongation. The partially smoothed prolongation variant is labelled adAFAC-JacP and the variant with partially smoothed restriction is labelled adAFAC-JacR. If a distinction is not made, then we refer to the variant with partially smoothed restriction exclusively. We verify that adAFAC-x solvers both converge and return a valid solution. The pure Poisson equation is a simple equation, but can become challenging for additive multigrid as the problem size increases. We can compute the analytical solution, therefore it is an effective test for solver consistency. Additional degrees of freedom and a large number of

grid levels causes overshooting in additive multigrid—even for the Poisson equation. Pure Poisson thus also provides a good test case for our adAFAC-x solver suite to show improved convergence rates for larger problem sizes.

We also explore possible stability improvements that adAFAC-x can bring. Additional tests, therefore, focus on non-homogeneous material parameters as these are known to introduce instabilities in some solvers. Large jumps in the material parameter introduce oscillations in the solution value for an additive solver—larger jumps corresponding to larger oscillations. Our adAFAC-x solver is designed to damp out such oscillations. The first discontinuous material parameter setup (E2) splits the domain into two equally sized sections. The domain boundary is axis orientated but not axis aligned. Although axis orientated, the split does not coincide with the mesh, as we employ three-partitioning of the unit square, therefore, this jump in material parameter cannot be accurately represented on any mesh level. This renders it a challenging solve and liable to introducing instabilities/oscillations in weaker solvers. The second discontinuous material parameter setup (E3) is no longer axis orientated. This exacerbates the same challenges seen with the first setup, and therefore a more powerful toolkit is required for fast convergence.

To analyse our asynchronous stencil assembly, we primarily use the latter of the discontinuous material parameter test setups. We do not investigate any setup with a homogeneous material parameter. For homogeneous setups, all stencils are uniform and known in advance so require minimal assembly. Our focus is on the more challenging of the two material parameter configurations, as it is both more expensive to assemble and will more clearly highlight any impact our lazy evaluation has on the stability of the solver. We require a large number of integration points, n, per cell around the discontinuity, but $n = 1$ will suffice in regions of constant ϵ. Localised significant changes to ϵ mean the choice of n is not uniform. The overall integration is therefore slow to converge due to the discontinuity. Additive multigrid tends to overshoot significantly when there are large material parameter changes, so we can readily expose any instabilities due to delayed assembly.

If tests are labelled as regular grid runs, each grid level is regular and, unless otherwise stated, we consequently end up with a mesh holding $(3^7-1)^d = 4,778,596$ degrees of freedom for $\ell_{max} = 7$. If tests are not labelled as regular grid runs, we rely on dynamic mesh refinement. Otherwise our experiments focus on $d = 2$ and start with a 2-grid algorithm ($\ell_{max} = 2$) where the coarser level has $(3-1)^d = 4$ degrees of freedom and the finer level hosts $(3^2-1)^d = 64$ vertices carrying degrees of freedom. From hereon, we add further grid levels and build up to an 8-grid scheme ($\ell_{max} = 8$).

When we apply adaptive mesh refinement to the setup with non-homogeneous boundary conditions (BC1), our code manually refines the cells along the bottom boundary in every other cycle, i.e. we refine the cells where one face carries $u|_{\partial\Omega} \neq 0$. The boundary is homogeneous in (BC2) so this is not required. We stop with this refinement when the current ℓ_{max} reaches a predetermined constant. Our manual mesh construction ensures that we kick off with a low total vertex count, while the solver does not suffer from pollution effects: The scheme kickstarts further feature-based refinement. Parallel to the manual refinement along the boundary, our implementation measures the absolute second derivatives of the solution along both coordinate axes in every single unknown. A bin sorting algorithm is used to identify the vertices carrying the (approximately) 10 percent biggest directional derivatives. These are refined unless they already meet ℓ_{max}. The overall approach is similar to full multigrid where coarse grid solutions serve as initial guesses for subsequent cycles on finer meshes, though our implementation lacks higher-order operators. All interpolation from coarse to fine meshes, both for hanging vertices and for newly created vertices, is d-linear.

7.1.3 Data measurements

Our runs employ a damped Jacobi smoother with damping $\omega = 0.6$ and report the normalised residuals

$$\frac{\|r^{(k)}\|_h}{\|r^{(0)}\|_h} \quad \text{where} \quad \|r^{(k)}\|_h^2 := \sum_i h_i^d (r_i^1(k))^2, \tag{7.1.1}$$

with k being the multigrid cycle count. $r_i^{(k)}$ is the residual in vertex i and h_i is the local mesh spacing around vertex i. Dynamic mesh refinement inserts additional vertices and thus might increase the number of entries in the residual vectors between two subsequent iterations, thereby increasing this metric. As a consequence, residuals under a Euclidean norm may temporarily grow due to mesh expansion. This effect is amplified by the lack of higher order interpolation for new vertices. The normalised residual (7.1.1) enables us to quantify how much the residual has decreased compared to the residual fed into the very first cycle.

The use of this norm follows on from statements about inner product norms in [13] and measurements obtained in [32]. This normalised residual is constructed as a discrete interpretation of the $L2$ norm. We assume that underlying shape functions are non-overlapping, therefore, weighting the squared residual from a vertex with the element size acts as a suitable approximation of the integral of the squared residual over that cell. Due to this weighting, residual contributions scale with the mesh—a large number of very fine grid cells doesn't dominate the norm due to their quantity—the norm incorporates contributions over the entire domain, so is an effective indication of global error. It is a bad indication if there exists only large local errors. If there is low error over the majority of the domain but a small region exhibits a large error, then the small region will only minimally effect the norm. As we are interested in tracking residual developments around discontinuities—which may exhibit large errors around the transition and low errors elsewhere—where appropriate, we also display the normalised maximum residual

$$\frac{max_i\{|r_i^{(k)}|\}}{max_i\{|r_i^{(0)}|\}}.$$

This second residual norm merely samples the single largest point in the domain in each iteration and acts as an identifier of localised errors. We track the progression of this sequence of largest values. which is analogous to the $LInf$ norm for a continuous space. This norm will be dominated by a single vertex that is slow to converge, or a single point exhibiting large oscillations in the solution value. Oscillations will most

probably occur near discontinuities. Sampling individual vertices therefore acts an early indicator of these instabilities.

7.1.4 Limitations of the current approach

For our current implementation we have targeted elliptic problems with discontinuous material parameters. Our asynchronous assembly process has been written with a finite element construction in mind, in particular one with simple basis functions with local support. This choice is a limit of the implementation, not of the algorithm, so our ideas are still applicable to more general problems. Our additive solver again targets the same problem types, though our current coarsening procedure, and thus choice of coarse grids and intergrid transfers, is somewhat dependent upon this choice of basis.

With this in mind, our scheme is well suited for solving random material parameter jumps; we can readily handle discontinuities of large magnitudes in the material parameter or random material parameters corresponding to Perlin noise like distributions. If the variation in the material parameter exists at a frequency longer than a fine grid cell, then it does not drastically impact the fine grid assembly process as the material parameter change per cell is still smooth. It will effect behaviour on the coarse grid however, where material parameter variations will be picked up and must be accurately represented. Our adAFAC-x damping is effective for these coarse grid variations: we damp out oscillations from the coarse grid due to long range variations. On the other side of things, the asynchronous stencil assembly is effective for large variations in the material parameters when they exist in the range of subcells of the fine grid. Asynchronous assembly initially constructs stencils which holds errors in the material parameter that belong to the same frequency/length as the fine grid cell itself. Using these stencils kickstarts solution convergence. The solution initially holds errors of the same frequency also. Subcell information is fed to the smoother in subsequent iterations. As long as the underlying discretisation is

stable and shows accurate representation eventually, then the solution will converge to the true solution. The asynchronous assembly simply provides a headstart. Our dynamic integration process and dynamical adaptivity means the discretisation will adapt to any material parameter configuration.

There exist problem types that our scheme is less well suited for; problems with anisotropic material parameters present their own set of challenges, for example. Additive multigrid (with geometric coarsening) is ill-suited for anisotropic problems. Stability can often be maintained, but solvers will exhibit very slow convergence. Due to our choice of geometric coarsening this still remains the case. We remain stable, and retain stability for a larger difference in material parameters, but converge very slowly. Although, we can construct operators very quickly due to our asynchronous assembly, so may appear well suited on the surface, we are unlikely to be the best choice due to our additive multigrid solver. Furthermore, many anisotropic problems use constant material parameters per dimension, which is trivial to construct using a tensor product formulation, so our asynchronous assembly would be overkill.

Our methodology also seems like a good fit for nonlinear problems. We use full solution representations on all levels, so could easily modify our solver to be a true FAS solver (rather than merely borrowing FAS ideas to make dynamical adaptivity less painful). Our asynchronous assembly continually re-integrates all equations to improve equation accuracy, which means we keep updating operator representation, so our solver adapts to changes in the local operator due to the changing local solution. However, vertical rippling of coarse grid operators slows the rate at which these changes are represented in coarse representations. Therefore, coarse grids use outdated solution representations in the operator, which may give rise to stability issues. These issues are flaws with our current implementation however, rather than the core tenets of asynchronous assembly.

We expect to see limited success when extending our solver to non-elliptic operators as the sole solver; for example, solving parabolic problems such as the Navier-Stokes equation. In its existing form, however, our solver could readily act as a sub-system

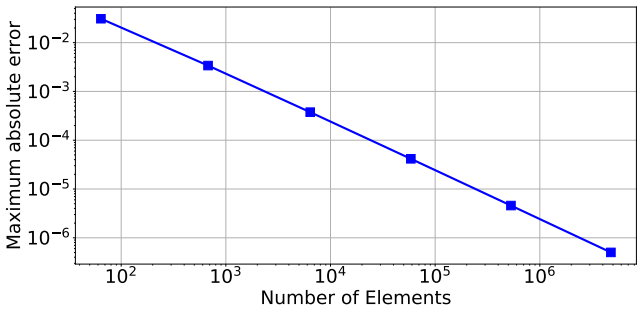

Figure 7.2: Mesh convergence for adAFAC-Jac as we increase the number of elements.

that acts upon elliptic subproblems within a larger equation. Struggling to tackle problems with different character, such as parabolic problems, is due to the nature of multigrid and its qualities as a solver—rather than an issue with our approach in particular. In its current form, our integration process has only been implemented for one case—the variable coefficient Poisson equation. By swapping out the exact function that performs the integration, so as to perform a different integration, we would be able to assemble a wider variety of matrices, and thus solve a wider variety of equations. Our solver would be able to handle this, it is the current implementation that holds it back.

7.2 adAFAC-x: Consistency

We initially focus on adAFAC-x and verify that it is consistent. That is, we check it is actually a valid solver. We initially highlight a quick mesh convergence study: we study the Poisson equation with a known solution. That is we record the maximum absolute error across the domain for the setup with RHS $f = -2\pi^2 sin(\pi x_0) sin(\pi x_1)$ and homogeneous boundary conditions (BC2). This has known exact solution $u = sin(\pi x_0) sin(\pi x_1)$. The plot of this can be seen in (Fig. 7.2). This confirms that as the mesh spacing decreases by a known factor—and therefore the total number of mesh elements increases by the square of that factor—the error decreases by square

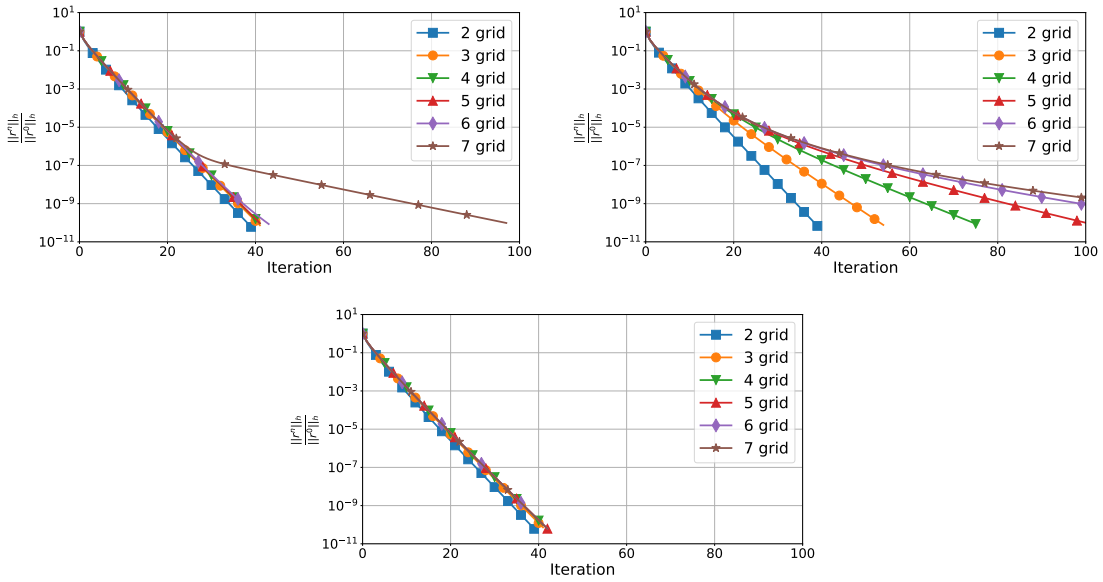

Figure 7.3: Solves of the Poisson equation on regular grids of different levels. We compare plain additive multigrid (top, left), multigrid using exponential damping (top, right), and adAFAC-Jac (bottom).

of that factor, i.e. we are in $O(h^2)$ (as expected from our second order accurate discretisation). This can be seen by the negative linear relationship between the number of mesh elements and the L^∞ error. Our first set of experiments then focus on the pure Poisson equation, i.e. our (E1) material parameter ($\epsilon = 1$ everywhere) and (BC1) (where $f = 0$). Multigrid is expected to yield a perfect solver for this setup: Each cycle (multiscale grid sweep) has to reduce the residual by a constant factor which is independent of both the degrees of freedom, i.e. number of vertices, and number of multigrid correction levels. Due to the constant material parameter, Ritz-Galerkin multigrid yields the same operators as rediscretisation, since BoxMG gives bilinear intergrid transfer operators for symmetric positive definite operators [118]. The setup is a natural choice to validate the consistency and correctness of the adAFAC-x ingredients. All grids are regular.

Our experiments (Fig. 7.3) confirm that additive multigrid converges insignificantly faster than the other alternatives for setups with a small number of meshes, if additive remains stable. However, the more grid levels are added, the more additive multigrid

	Degrees of Freedom					
	64	676	6,400	58,564	529,984	4,778,596
Additive Multigrid						
Total runtime, [s]	1.97E-01	1.06E+00	6.07E+00	5.65E+01	5.31E+02	9.85E+03
Average time per solver iteration, [s]	4.18E-03	2.05E-02	1.19E-01	1.09E+00	9.75E+00	9.00E+01
Assembly time, [s]	2.97E-02	1.95E-01	1.07E+00	1.07E+01	1.02E+02	1.02E+03
adaFAC-JacR						
Total runtime, [s]	2.29E-01	9.89E-01	6.49E+00	5.99E+01	5.39E+02	5.31E+03
Average time per solver iteration, [s]	4.43E-03	1.89E-02	1.29E-01	1.15E+00	1.02E+01	9.96E+01
Assembly time, [s]	5.05E-02	1.93E-01	1.07E+00	1.05E+01	1.00E+02	1.02E+03
adaFAC-JacP						
Total runtime, [s]	2.31E-01	1.08E+00	6.65E+00	6.19E+01	5.60E+02	5.38E+03
Average time per solver iteration, [s]	4.56E-03	2.06E-02	1.30E-01	1.19E+00	1.07E+01	1.02E+02
Assembly time, [s]	4.74E-02	1.94E-01	1.07E+00	1.06E+01	1.01E+02	1.00E+03
adaFAC-PI						
Total runtime, [s]	1.97E-01	9.70E-01	6.22E+00	5.69E+01	5.12E+02	4.98E+03
Average time per solver iteration, [s]	4.19E-03	1.84E-02	1.19E-01	1.10E+00	9.80E+00	9.45E+01
Assembly time, [s]	2.98E-02	1.93E-01	1.23E+00	1.06E+01	1.00E+02	1.00E+03
Exponentially Damped						
Total runtime, [s]	2.10E-01	1.18E+00	1.01E+01	1.21E+02	1.35E+03	1.49E+04
Average time per solver iteration, [s]	4.22E-03	1.78E-02	1.19E-01	1.09E+00	9.79E+00	9.31E+01
Assembly time, [s]	4.01E-02	1.98E-01	1.13E+00	1.04E+01	9.97E+01	1.01E+03

Table 7.2: A breakdown of time-to-solution, average time taken per iteration, and total assembly cost for a selection of our additive solvers as we increase the total degrees of freedom used in the solve when solving for the Poisson equation on a regular grid.

overshoots per multilevel relaxation. When we add a seventh correction level, this suddenly makes the plain additive code's performance deteriorate. With an eighth level added, the solver would diverge (not shown). Exponentially damped multigrid does not suffer from such an instability for a large number of levels, but the aggressive damping of coarse grid influence prevents the solution updates from retaining a long-range impact, so updates do not propagate quickly. The convergence speed suffers from additional degrees of freedom and additional correction levels. Here, we only show the graph for adAFAC-Jac with smoothed restriction—our other damping setups produce visually identical graphs. All three of our damping parameter choices for adAFAC-x are stable, but they do not suffer from the speed deterioration of an exponentially damped scheme—they retain textbook multigrid convergence. Their localised damping makes both schemes effective and stable. A breakdown of time-to-solution and the time of differing phases is shown in Table 7.2. We note that for the same mesh size the time for an individual iteration is consistent between additive multigrid and our adAFAC-x setups, as is total assembly time. In situations where undamped additive multigrid converges, we see similar rates of convergence in our

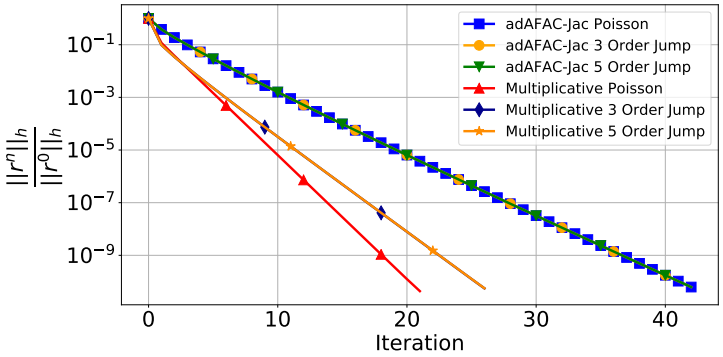

Figure 7.4: Comparing the number of iterations our adAFAC-Jac solver requires to converge against a multiplicative multigrid solver provided by PETSc. Both cases solved for approximately $4 \cdot 10^7$ degrees of freedom on regular grids for different material parameter setups.

damped implementations to that of undamped additive multigrid. Therefore, there is no appreciable increase in computational cost when using auxiliary damping.

We now offer a brief comparison between our damped additive solver and an effective existing multiplicative solver, the multiplicative multigrid solver provided within PETSc [119]–[121]. This shows how our damping methodology compares to an actual multiplicative solver, rather than just the advantage and improved stability our damping scheme brings to an additive scheme. We do not use a Krylov solver on any level and stick to damped Jacobi smoothers across all levels using a $V(1,1)-$cycle. This is not a realistic way to use PETSc but serves as a better comparison to our damped additive scheme. We run a PETSc solver on a regular grid with $4,198,401$ degrees of freedom, the regular grid for our damped additive scheme, on the other hand, is slightly larger, and has $4,778,596$ degrees of freedom. Experimental runs stick to setups with zero right hand side and inhomogeneous boundary conditions (BC1), and solve for either constant material parameter (E1, labeled as Poisson), or the setup with a single jump in the middle of the domain (E2) with jumps of either three or five orders of magnitude. The results of these experiments can be seen in (Fig. 7.4). Unsurprisingly, the multiplicative solver outperforms our damped additive scheme and converges in roughly half the number of iterations for

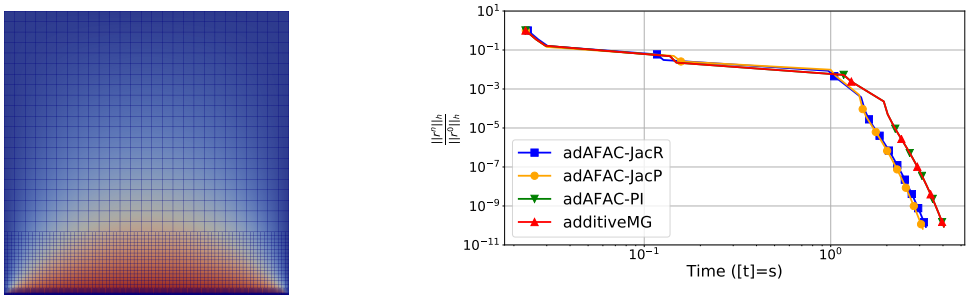

Figure 7.5: Left: Typical adaptive mesh for pure Poisson (constant material parameter) once the refinement criterion has stopped adding further elements. Right: We compare different solvers on the pure Poisson equation using a hybrid FMG-AMR approach starting at a two grid scheme and stopping at an eight grid

all the material parameter configurations. This is to be expected, considering the additional smoothing steps in the PETSc setup. PETSc performs a full $V(1,1)-$cycle rather than merely approximating a level specific $V(1,0)-$cycle. We argue that the increased potential for concurrency within an additive scheme means that our damping scheme remains competitive. Also of note, when using the built-in additive solver in PETSc on this setup it fails to converge, so the lack of stability for vanilla additive multigrid can clearly be seen.

Despite the instability of plain additive multigrid, we continue to benchmark against the undamped additive scheme, as exponential damping is both not competitive and not unambiguously defined for adaptively refined meshes. We start to investigate schemes with dynamic feature based refinement on ragged grids. Experiments from hereon now may be reasonably irregular/coarse to circumnavigate the instabilities. Feature-based dynamic refinement criterion makes the mesh spread out from the bottom edge when $u|_{\partial\Omega} = sin(\pi x_0)$ (Fig. 7.5). We start from a small regular grid with a single correction level, and add additional points throughout the solver's run-time. Additional points are added to resolve features of interest within the solution. To assess its impact on cost, we count the number of required degrees of freedom updates plus the updates on coarser levels. These degree of freedom updates correlate directly to run-time.

One smoothing step on a regular mesh of level eight yields $4.3 \cdot 10^7$ updates plus the updates on the correction levels. If the solver terminated in 40 cycles, the number of cycles required earlier for convergence, we would have to invest more than 10^9 updates per solve. Dynamic mesh unfolding reduces the total required updates to reduce the residual by up to three orders of magnitude. The final mesh now holds fewer than 2000 degrees of freedom. This corresponded to a run time of approximately 3 seconds for a single core study. For Poisson, this saving applies to both plain additive multigrid—while it remains stable—and our adAFAC-x variants.

If ran with BoxMG, our codebase uses Ritz-Galerkin coarse operator construction for both the correction terms and the auxiliary coarse grid operators in adAFAC-Jac. We validated that both the algebraic intergrid transfer operators and geometric operators yield exactly the same outcome. This is correct for pure Poisson as BoxMG yields geometric operators here, therefore the use of Ritz-Galerkin coarse operator construction for the correction terms yields the same result as rediscretisation.

Observation 1. *Our code is consistent. For very simple, homogeneous setups, however, it makes only limited sense to use adAFAC-x—unless there are many grid levels. If adAFAC-x is to be used, adAFAC-PI is sufficient. There's no need to explicitly construct an actual additional auxiliary equation.*

We conclude with the observation that all of our solvers, if stable, converge to the same solution. They are real solvers, not mere preconditioners that only yield approximate solutions.

7.3 adAFAC-x: Stability

7.3.1 Regular grid with one material jump

We next study a setup where the material "jumps" in the middle of the domain (E2). The stronger the material transition, the more important it is to reflect the

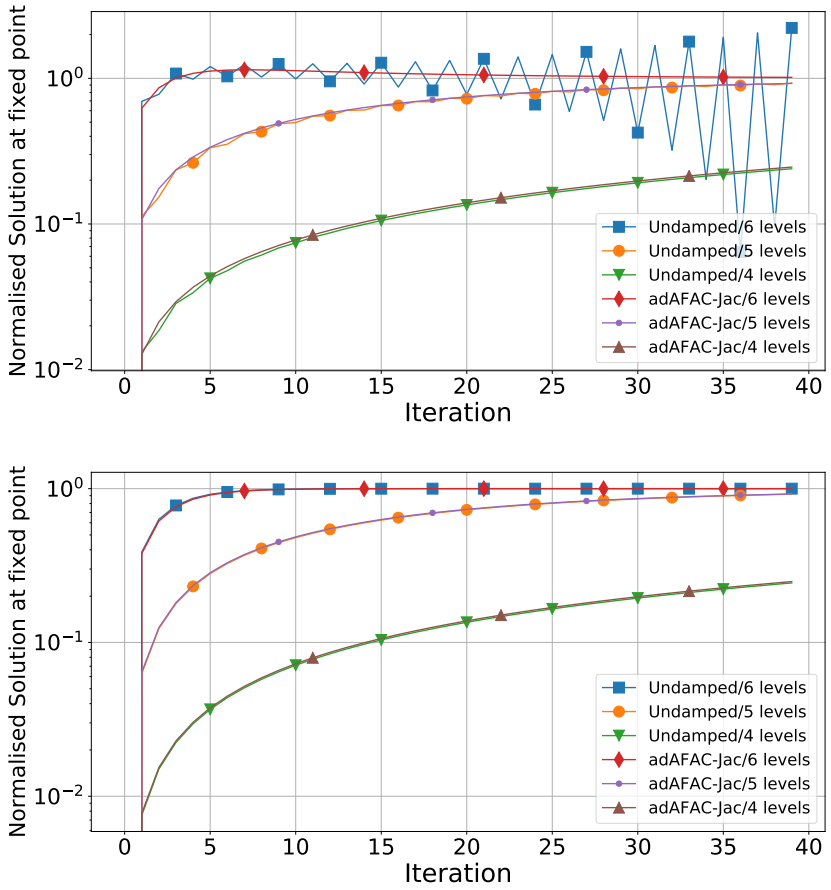

Figure 7.6: The domain material is split into two halves with an ϵ jump from $\epsilon = 1$ to $\epsilon = 10^{-7}$. Solution development in sample point next to a discontinuity, normalised by the true solution value at that point, i.e. one means the correct value. We compare d-linear intergrid transfer (top) to BoxMG operators (bottom).

ϵ changes in the intergrid transfer operators. Otherwise, a prolongation of coarse grid corrections introduces errors close to $x_1 = 0.5$. Dynamic refinement yields many additional degrees of freedom around this discontinuity; however, due to the fact the space-tree setup uses three-partitioning, the discontinuity is never resolved exactly. As no grid in the present setup has degrees of freedom exactly on the material transition, the intergrid transfer operators are also never able to mirror the material transition exactly. This extends to the coarse grid levels—they are unable to exactly represent the chosen material jump and thus solve an inaccurate equation. The additional fine grid vertices from AMR therefore act as a fix on the inaccurate

coarse grid solve. This ideology can be viewed as a foil to the damping used within adAFAC-x. adAFAC-x removes coarse grid errors by improving the solve we perform on the coarse, so coarse corrections are more accurate. Adaptivity removes coarse grid errors by adding finer elements along the transition. Rather than simply using adaptivity to fix errors we make errors smaller right from the start. The coarse grid corrections incorporate additional fine grid information to produce corrections with less error.

Without dynamic adaptivity, multigrid runs the risk of deteriorating in the multiplicative case and becoming unstable in the additive case. To document this phenomenon, we monitor the solution in one sample point coinciding with the real degree of freedom next to $x = (0.5, 0.5)^T$, and employ a jump in ϵ of seven orders of magnitude. We restrict our focus to the boundary setup (BC2). A regular grid corresponding to $\ell = 6$ is used. We start from a single grid algorithm, and add an increasing number of correction levels. Not all possible grid level setups are shown here. Without algebraic intergrid transfer operators, oscillations arise if we do not use our additional damping parameter (Fig. 7.6). For undamped additive multigrid, oscillations increase as the number of coarse grid levels used increases. Additive multigrid is stable in cases with fewer than six grid levels, but it shows a reduced rate of convergence. When only four grid levels are used it takes 15 iterations for the solution approximate to be within an order of magnitude of the true solution, for setups with five and six grid levels this is achieved after only one iteration. Undamped additive multigrid with six levels shows oscillations in solution value from the very beginning. The oscillations increase in size with additional solver iterations. Our damping parameter eliminates these oscillations and does not harm the rate of convergence. adAFAC-Jac converges for the six level setup and for the four and five level setups both undamped and damped additive multigrid see almost visually identical curves and rates of convergence. Algebraic intergrid transfer operators eliminate these oscillations, too. Both additive multigrid and adAFAC-Jac do not show oscillations when using BoxMG intergrid transfer operators and when using the

same number of grid levels show similar rates of convergence. The results show why codes without algebraic operators and without damping usually require a reasonably coarse mesh to align with ϵ transitions.

7.3.2 Adaptive mesh refinement with one material jump

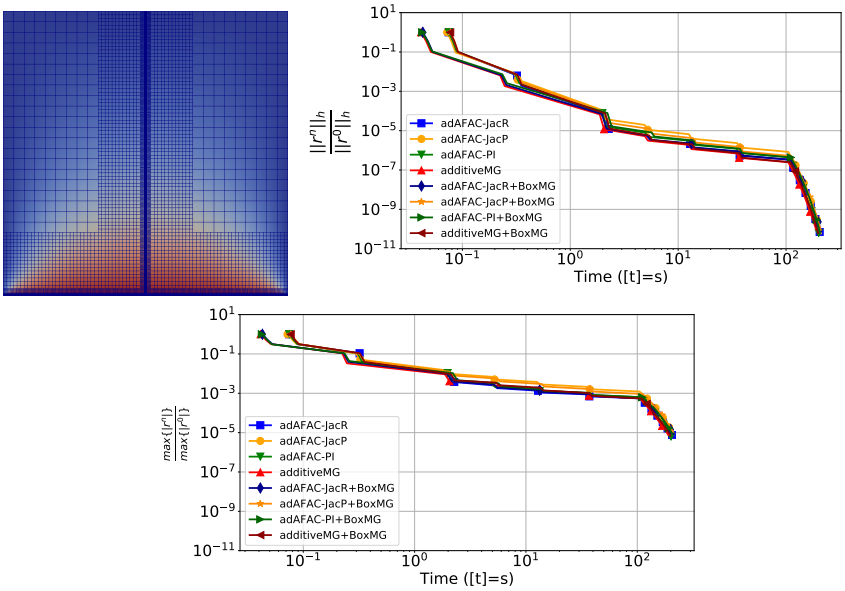

Figure 7.7: Top Left: The domain material is split into two halves with an ϵ jump from $\epsilon = 1$ to $\epsilon = 10^{-k}$. Typical adaptive mesh for single discontinuity setup once the refinement criterion has stopped adding further elements. Top Right: $\epsilon \in \{1, 10^{-1}\}$, i.e. the material parameter changes by one order of magnitude. We present only data for converging solver flavours. Bottom: The same setup but for the normalised maximum norm.

We continue with our analysis of a single material jump (E2) but switch to dynamically adaptive meshes. All experiments use the already detailed AMR/FMG setup, i.e. start from a coarse regular mesh and then dynamically adapt the grid. As these runs also used non-homogeneous boundary conditions (BC1), we observe that the hard-coded grid refinement refines along the stimulus boundary at the bottom, while the dynamic refinement criterion unfolds the mesh along the material transition (Fig. 7.7).

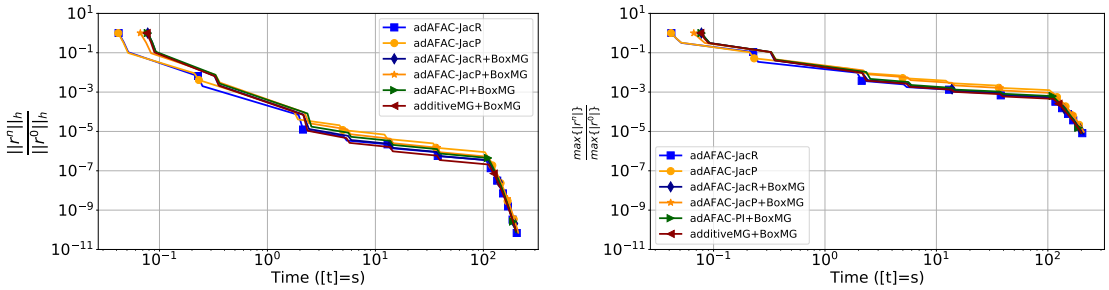

Figure 7.8: Setup of Fig. 7.7 but with a five orders of magnitude jump in the material parameter. We present only data for converging runs and observe that fewer solver ingredient combinations converge.

For a single order of magnitude jump in ϵ, all showcased multigrid variants are stable. The residual plot in the maximum norm validates our statement that large errors arise along the material transition when we insert new degrees of freedom. The absence of a higher-order interpolation for new degrees of freedom hurts, in that it harms the overall convergence speed, but it does not destroy the overall stability. We currently perform a "wrong" interpolation when setting up the fine grid, so must perform at least one fine grid smoothing step to correct. Once the dynamic AMR stops inserting new vertices—this happens once the grid holds $1.3 \cdot 10^6$ many vertices and after almost 10^6 degrees of freedom have been processed in total—the residual drops under both norms. This is an increase relative to the Poisson equation—of around a hundred times—but as there are now a hundred times as many vertices, this is still competitive. A large part of the computational time is due to equation reconstruction performed after refinements.

Once we increase the size of the changes in ϵ (to a jump of two orders of magnitude), undamped additive multigrid with geometric intergrid transfer operators fails to converge. Using BoxMG, however, restores stability. We need an algebraic interpolation routine, or alternatively one of our adAFAC-x variants. adAFAC-Jac with bilinear transfer operators converges for all $\epsilon = 10^{-k}$ values tested. This is true when either smoothed restriction or prolongation is used for the auxiliary grids. However, undamped additive multigrid and adAFAC-PI fail to converge without

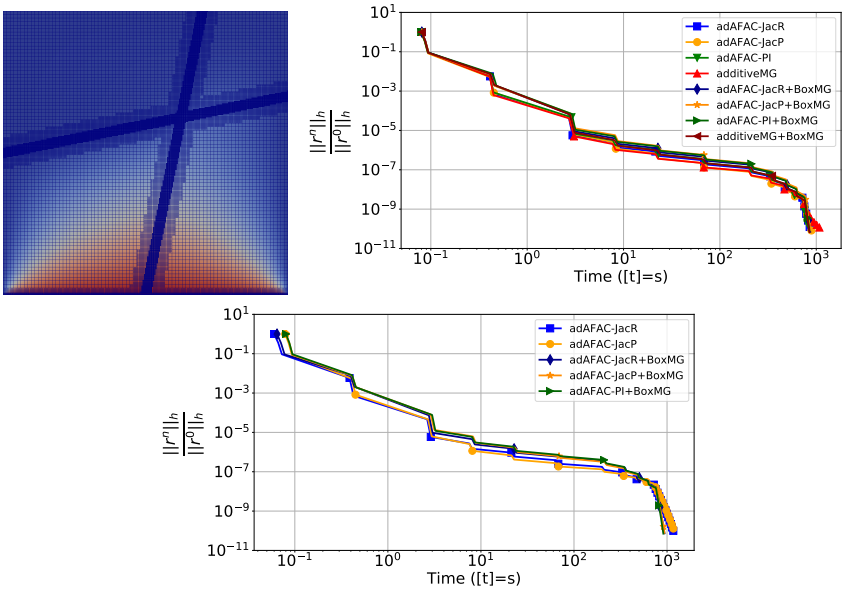

Figure 7.9: Typical adaptive mesh for a setup where the regions with different material parameter ϵ are not axis-aligned. One order of magnitude differences in the material parameter (top right) vs. three orders of magnitude (bottom).

BoxMG once the ϵ-transition becomes too harsh (Fig. 7.8). The geometric intergrid transfer approach suffers from oscillations around the material transition. All stable solvers play in the same league.

Observation 2. *If we face reasonably small jumping materials, adAFAC-PI is superior to plain additive multigrid, adAFAC-Jac or any algebraic-geometric extension, as it is both stable and simple to compute. Once the jump grows, adAFAC-Jac becomes the method of choice. Its auxiliary damping equations compensate for the lack of algebraic intergrid transfer operators, which are typically not cheap to compute.*

7.3.3 Adaptive mesh refinement with non axis-aligned subdomains

We move on to our experimental setup with a deformed checkerboard setup (E3). Similar to the previous setup (E2), the material transitions are again not axis aligned, but in addition, they are now also skew. This is harder to represent in coarse grid

equations. Due to our use of space-trees, the discontinuity does not exactly bisect cells, so fine grid equations over a discontinuity no longer follow a fixed, repeating pattern. We can never exactly resolve the material transitions with the degrees of freedom or intergrid transfer operators. We again focus on (BC1), so a homogeneous right-hand side and non-homogeneous boundary conditions (Fig. 7.9), where the dynamic adaptivity criterion unfolds the mesh along the material transitions. The solution behaviour within the four subregions itself is smooth, i.e. diffusive, and the adaptivity around the material transitions thus is wider, more balanced, than the hard-coded adaptivity directly at the bottom of the domain.

With small variations in ϵ, this setup does not pose a challenge to any of our damped solvers, irrespective of whether they work with algebraic or geometric intergrid transfer operators. Additive multigrid with d-linear grid transfer operators does not retain textbook multigrid convergence for the full solve, however. Once grid refinements cease all damped setups show this rate of convergence. Without BoxMG operators, additive multigrid stagnates slightly—it requires 10^7 solution updates to converge. This increase in required updates and iteration does not drastically increase the runtime relative to other solvers however. This is an order of magnitude slower than the number of updates required for the Poisson equation. There are oscillations in the solution that slow the rate of convergence—for low ϵ jumps these are slowly removed (hence the stagnation), but once ϵ becomes large the oscillations increase and cause divergence. Case in point: we look to the setup with increasing large jumps in ϵ. For $\epsilon = 3$, we observe that additive multigrid starts to diverge, even with algebraic grid transfers and hence is not shown on the plot. Smooth regions are still sufficiently dominant and we suffer from overcorrection between them. adAFAC-PI performs better yet requires algebraic operators to remain robust up to ϵ variations of three orders of magnitude. With geometric operators both options for adAFAC-Jac remain stable for all studied setups, up to and including the five order of magnitude jump. adAFAC-Jac with algebraic operators consistently outperforms its geometric cousin. BoxMG's accurate handling of material transitions

	Sequential Code	Shared Memory Code (Number of Cores)					
		1	2	4	6	8	12
Runtime [s]	9.41E+03	1.14E+04	6.19E+03	3.79E+03	2.71E+03	2.34E+03	1.93E+03
Total Instruction Calls	3.05E+12	3.67E+12	3.58E+12	3.62E+12	3.65E+12	3.80E+12	3.83E+12
L1 request rate	6.99E-02	8.04E-02	7.84E-02	7.94E-02	7.99E-02	8.32E-02	8.33E-02
L1 miss rate	4.19E-05	3.00E-04	3.00E-04	3.00E-04	3.00E-04	5.00E-04	6.00E-04
L2 request rate	2.00E-03	2.16E-02	1.30E-02	1.46E-02	1.52E-02	1.52E-02	1.85E-02
L2 miss rate	7.00E-04	3.90E-03	3.30E-03	3.70E-03	3.90E-03	4.00E-03	4.70E-03

Table 7.3: Analysis of performance for our adAFAC-Jac solver. We compare the performance of the sequential implementation of our code with the shared memory implementation for a regular grid with $4,778,596$ degrees of freedom.

decouples the subdomains from each other on the coarse correction levels. Updates in one domain thus do not pollute the solution in a neighbouring domain.

Observation 3. *While the auxiliary equations can replace/exchange algebraic operators in some cases, they fail to tackle material transitions that are not grid-aligned.*

7.4 adAFAC-x: Performance

We perform an initial performance analysis of our code, where we verify that our implementation is single-touch: that is, we check that we see minimal cache misses. An illustration of hardware measurements and timings can be seen in Table 7.3. These initial tests are for a regular grid with over $4 \cdot 10^6$ DoFs and a constant material parameter. We report cache requests and misses as a rate, i.e. a proportion, rather than as absolute values, and note that we see similar excellent cache utilisation to that seen in Reps and Weinzeirl's implementation [32]. Again, we reiterate their insights, that due to the single-touch policy per unknown and the localised traversal of the space-tree we can assume that each unknown is only loaded into the cache once per grid traversal. The low rate of cache misses, for both L1 and L2 caches, confirms this hypothesis. We also note that we see reasonable scaling (i.e. a reasonable reduction it time-to-solution) as we increase the number of cores. Although we see an initial increase in time-to-solution due to the TBB overhead for the single core setup.

We close our adAFAC-x focused experiments with a scalability exercise. Both the

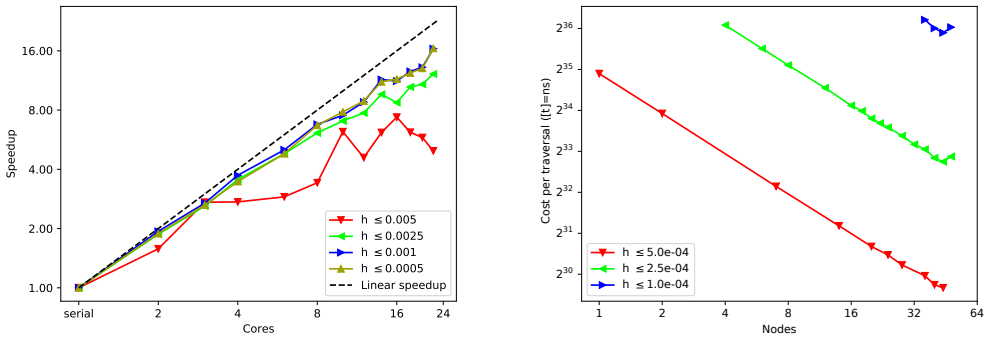

Figure 7.10: Left: Shared memory experiments with adAFAC-x. All solver variants rely on the same code base, i.e. exchange only operators, such that they all share the same performance characteristics. Right: Some distributed memory run-time results with the time for one multiscale grid sweep. This corresponds to one additive cycle as we realise single-touch semantics. We study three different mesh sizes given via upper bounds on the h. Two ranks per node, i.e. one rank per socket, are used.

shared and the distributed memory parallelisation of our code use a multilevel non-overlapping domain decomposition [44]. The non-overlapping domain decompositions are derived using a space-filling curve: The fine grid cells are arranged along the Peano space-filling curve and cut into curve segments of roughly the same number of cells. We use a non-overlapping domain decomposition on the finest mesh. Logically, our code does not distinguish between the code's shared and distributed memory strategy. They both decompose the data in the same way. The distributed memory variant, however, replaces memory copies along the boundary with MPI calls. All timings rely on run-times for one cycle of a stationary mesh, i.e. load ill-balances and overhead induced by adaptive mesh refinement are omitted.

For all experiments, we start adAFAC-x and wait until the dynamic adaptivity has unfolded the grid completely such that it meets our prescribed h as a maximum mesh size. The associated total number of degrees of freedom are: 4,778,596 for $h \leq 5.00E-04$; 43,033,600 for $h \leq 2.50E-04$; and 387,381,124 for $h \leq 1.00E-04$. We use two MPI ranks per node (one rank per socket) and use a shared memory

splitting of elements within a node. Our splitting strategy assigns the same number of degrees of freedom to each core/thread—results were ran on a static, regular grid so this can be guaranteed. We furthermore hard-code the domain decomposition such that the partitioning is close to optimal: We manually eliminate geometric ill-balancing, and we focus on the most computationally demanding cycles of a solve. Cycles before that, where the grid is not yet fully unfolded, yield performance which is similar to experiments with a bigger h.

Our shared memory experiments (Fig. 7.10) show reasonable scalability up to eight cores, if the mesh is detailed. The curves are characteristic for both adAFAC-PI and both variants of adAFAC-Jac, i.e. we have not been able to distinguish the run-time behaviour of these approaches. If the mesh is too small, we see strong run-time variations. Otherwise, the curves are reasonably smooth. Overall, the shared memory efficiency is limited by less than 70% even if we make the mesh more detailed.

Our code employs a very low order discretisation and thus exhibits a low arithmetic intensity. This intensity is increased by both adAFAC-PI and adAFAC-Jac, but the increase is lost behind other effects, such as data transfer or the management of adaptive mesh refinement. The reason for the performance stagnation is not clear, but we may assume that NUMA effects play a role, and that communication overhead affects the run-time, too. With a distributed memory parallelisation, we can place two MPI ranks onto each node and use shared memory parallelisation within the node. NUMA then does not have further knock-on effects, and we obtain smooth curves until we run into too small partitions per node. With a low-order discretisation, our code is communication-bound—in line with most multigrid codes—yet primarily suffers from a strong synchronisation between cycles.

Due to a non-overlapping domain decomposition on the finest grid, all traversals through the individual grid partitions are synchronised with each other. Our adAFAC-x implementation merges the coarse grid updates into the fine grid smoother, but each smoothing step requires a core to synchronise with all other cores. We

eliminate strong scaling bottlenecks due to small system solves, but we have not yet eliminated scaling bottlenecks stemming from a tight synchronisation of the (fine grid) smoothing steps.

Observation 4. *Despite adAFAC-x's slight increase of the arithmetic intensity, it seems to be mandatory to switch to higher order methods [122] or approaches with higher asynchronicity [90] to obtain better scalability. This is in line with other research.*

7.5 Delayed stencil integration: Consistency

We now move on to experiments studying our asynchronous stencil assembly. Specifically we investigate consistency with dynamic termination criteria and the possibility of starvation effects. We largely focus on the deformed checkerboard material parameter setup (E3) seen in the previous section with the fine grid hosting a homogeneous right-hand side and non-homogeneous boundary conditions (see Fig. 7.1). The discontinuities are not axis aligned or orientated, therefore accurately capturing them within a fine grid cell integration will require a large number of integration points. This is an expensive integration and a good use case for our asynchronous assembly—the assembly process is expensive so we will clearly see any possible gains. Moreover, we have shown this to be a challenging material parameter for a solver. If delayed stencil integration introduces any instabilities to the solver then they will be readily apparent. Again, we initially study consistency, before investigating AMR applications. We begin this section of experiments with some studies on dynamic termination criteria. Most codes terminate the solve as soon as the normalised residual runs under a given threshold or stagnates. If we use delayed, asynchronous stencil integration, i.e. we do not wait per cell for the underlying next step of the integration to terminate, we thus run into the risk that we terminate the solve prematurely, i.e. before the correct local assembly matrices have been computed. From

an assembly point of view, this is a starvation effect: The assembly tasks are issued, yet have not been scheduled and thus cannot affect the solve. We end up with the solution to a "wrong" problem described by these inaccurate operators.

We investigate this hypothesis simulating our test equation with a high parameter variation on a regular Cartesian mesh hosting 59,049 degrees of freedom. We initially focus on geometric intergrid transfer operators and a material parameter jump of five orders of magnitude, using the non-axis aligned regions seen in Fig. 7.1. Four multigrid correction levels are employed. Our experiment tracks both the residual development and the number of background tasks pending in the ready queue. We reiterate that they are issued with low priority such that the incremental improvement of the assembly process does not delay the solver iterations.

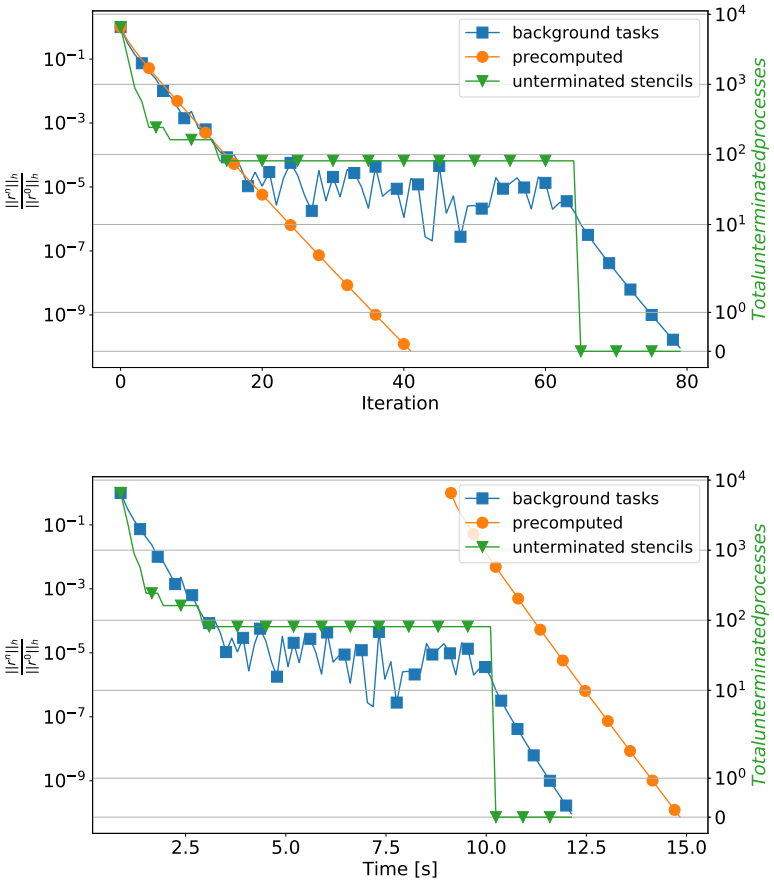

Figure 7.11: Convergence of delayed operator evaluation vs. precomputed stencils/operators per iteration (top) and against real time (bottom).

For smooth ϵ distributions, we have not been able to spot any deterioration of the residual evolution due to the delayed stencil integration (not shown). For rapidly changing ϵ, e.g. changing regions of $\epsilon = 1$ or $\epsilon = 10^{-5}$, the solver's behaviour changes dramatically however (Fig. 7.11). Using a delayed assembly (stencil computation) deployed to background tasks, the solver iteration count required to reduce the normalised residual to a factor of 10^{-10} has doubled compared to a solve where all operators are accurately computed prior to the multigrid solver iterations beginning. Initially, both methods show a similar rate of convergence; however, the delayed solve soon enters a regime where its residual almost stagnates around 10^{-5}. Throughout the initial residual decay, the number of pending background tasks reduces dramatically. While the residual stagnates, the number of background integration tasks remains constant, however. Towards the end of the residual plateau, the number of background tasks drops to zero and the solver recovers and exhibits multigrid performance again.

Our solver spans one background assembly task per fine grid cell initially and continues to work with a geometric approximation to the local assembly matrix from thereon. Most of the assembly tasks are associated with cells covering smooth ϵ distributions. Thus, they discover that the assembly approximation is sufficiently accurate almost immediately, i.e. after increasing the number integration points per cell n once. They terminate and do not reschedule any tasks for this particular cell. Only the few tasks associated with regions close to the significant ϵ variations require repeated rescheduling while increasing n. By the time only these rescheduled tasks remain, the lack of accurate subcell material representations for some cells becomes detrimental to the rate of convergence. We reach a point where the current solution accuracy is balanced with the error of the stencils/assembly matrices that still have to be integrated properly. Updates to the cell matrices hence "introduce" error—or rather expose errors in the solution that the previously held stencil was unable to account for. Due to the elliptic nature of the operator, these errors spread through the entire domain. The entire solver stalls. At the point all the background

integration tasks have converged, i.e. do not reschedule themselves anymore, we regain multigrid convergence as we finally solve the correct system that no longer changes.

We note two key observations from this:

Observation 5. *The fine grid changing and equations rippling to the coarse grids can prevent the residual from continually converging/decreasing. It introduces oscillations.*

and

Observation 6. *Dynamic termination criteria for the equation system solver also have to be designed carefully with delayed operator assembly, as the solver might converge towards a wrong solution.*

While our convergence considerations therefore seem to not favour the delayed, asynchronous assembly, if we instead make a comparison with regards to the execution time, we change the picture (Fig. 7.11). An increased iteration count in the solver is negated due to the headstart the delayed evaluation gives the solver. The setup also highlights, that for certain solves, precomputing accurate stencils can take a greater amount of time than the solve itself. Finally, we see that the time-to-solution of the delayed assembly is superior compared to the explicit a priori assembly. As we kick off with low-accuracy operators, we effectively merge the first few multigrid cycles with the actual assembly process. This confirms the illustration which we initially presented in Fig. 4.5 actually holds. For this simple setup, the point in time at which delayed evaluation has computed an accurate solution representation is a similar point in time to that when the precomputed stencil has computed an accurate stencil; even though our precomputation routines employ a dynamic n choice as well. Therefore the delayed method can be seen as a way of computing a reasonably accurate initial guess, and the delayed assembly manages to maintain the lead from its headstart.

We further confirm that this gain for lazy stencil integration holds for larger grids and present a selection of run-times for a regular grid with six multigrid levels and 529,984 degrees of freedom (Fig. 7.4). Here we show experimental data for completed solver runs for the skew checkerboard setup (E3) with a jump of three orders of magnitude ($k = 3$) and non-homogeneous right-hand side (BC1) using BoxMG intergrid transfers. We report walltime—the total time taken for the assembly phase and the multigrid iterations—for a solve that assembles all operators a priori and the speedup gained from using lazy stencil integration. For the three multigrid solvers we present experimental data for—adAFAC-Jac, adAFAC-PI and vanilla additive multigrid—we see an improvement in time-to-solution of a factor of 2/3 (which corresponds to a speedup of 1.5). This proportional reduction in total run-time remains roughly constant as we increase the total cores used in a solve—more cores used in the solves already reduces time-to-solution. This further highlights the reduction in overall run-time of our approach.

We provide a further comparison for the same experimental material parameter setup but instead with geometric intergrid transfer operators. Like in Table 7.4, we again compare for the solve with $529,984$ degrees of freedom, but also compare for a larger setup with $4,778,596$ degrees of freedom. We again provide walltime but further break the runtime down into phases: the assembly (grid setup) time and the solve time itself. This split should highlight more explicitly the performance benefit of asynchronous assembly. This can be seen in Table 7.5. For both mesh sizes, we see that asynchronous assembly dramatically reduces the time taken for the initial assembly phase—this is expected as we are performing the same grid construction and data structure setup as with an a priori assembly but the asynchronous setup only performs a less expensive/accurate stencil integration. For both meshes, the assembly phase with asynchronous assembly takes approximately a third of the time it would have otherwise taken. For the solver phases, both a priori assembly and asynchronous assembly take the same number of multigrid solver iterations to converge to the same accuracy. Asynchronous assembly does take slightly more time

Solver Type	Run-time [s] / Speedup			
	2	4	8	12
adAFAC-Jac: *Original run-time*	847.90	491.44	297.63	273.34
Speedup	1.54	1.50	1.50	1.604
adAFAC-PI: *Original run-time*	864.16	503.31	293.83	291.64
Speedup	1.49	1.48	1.40	1.49
Additive: *Original run-time*	976.39	562.21	323.51	279.31
Speedup	1.68	1.62	1.61	1.63

Table 7.4: Total solver timings for BoxMG including all assembly time. The first row in each denotes the time-to-solution with a precise a priori assembly, the second the speedup obtained through lazy integration.

adAFAC-Jac	Total Time [s]	Assembly Time [s]	Iteration Time [s]	Average Time Per Solver Iteration [s]	Total Iterations
Precompute					
529,984 DoFs	550.47	222.35	328.12	6.08	54
4,778,596 DoFs	5,736.45	2,209.67	3,526.78	57.82	61
Asynchronous Assembly					
529,984 DoFs	481.52	83.09	398.43	7.38	54
4,778,596 DoFs	4,947.06	757.62	4,189.45	67.57	62

Table 7.5: Total solver timings when using geometric intergrid transfer operators, across multiple large discontinuities, including all assembly time for two different mesh sizes. The first row in each section denotes the time-to-solution with a precise a priori assembly, the second the speedup obtained through lazy integration.

per iteration however. This is both due to computing the stencil integrations within the solver iterations, which requires compute time, and having to recompute coarse grid stencils each solver iteration. There is a noteworthy increase on the runtime of each individual iteration, but not enough that it increases the runtime of the solve in total. That is, we see an overall reduction in time-to-solution.

Observation 7. *A delayed operator integration pays off in time-to-solution for rough material parameters.*

7.6 Delayed stencil integration: Stability

7.6.1 Robustness and iteration counts on a regular grid

To test whether additional instabilities are introduced due to our laziness, we again perform consistency studies on a regular grid. We start with regular grids, specifically

we again test with a grid setup that holds a total of $4,778,596$ degrees of freedom on the fine grid. Again using (E2), the non-axis aligned geometric parameter distribution from Fig. 7.1, we fix $\epsilon = 1$ in the bottom left and top right subdomain and make $\epsilon = 10^{-k},\ k \in \{1, 2, \ldots, 5\}$ otherwise. The total number of material parameter sampling points n within a numerical integration is two for cells that do not lie on the discontinuity. Most cells that hold a discontinuity required 20 sampling points All data report normalised residuals, i.e. the residual development in the discrete $L2$ norm relative to the initial one. We stop when the initial residual is reduced by ten orders of magnitude.

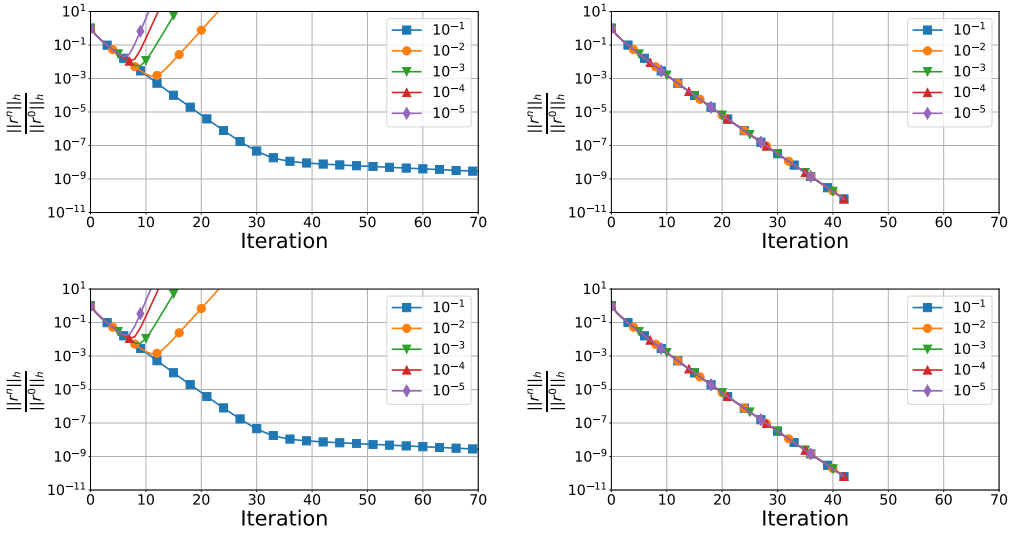

Figure 7.12: Number of iterations until convergence is reached for a selection of setups using geometric intergrid transfers, with checkerboard material parameter as the size of the jump increases. We either use an undamped setup (left) or our damped setup adAFAC-Jac (right). We compare the conventional method of precomputing all operators before the first solver iteration (top) to our delayed stencil integration with vertical rippling (bottom).

Our benchmarks compare five methods of initialising the stencils against each other:

- Exact integration + Geometric transfers: An exact computation of all fine grid is computed and all coarse grid operators using geometric intergrid transfer operators are setup prior to the first multigrid cycle.

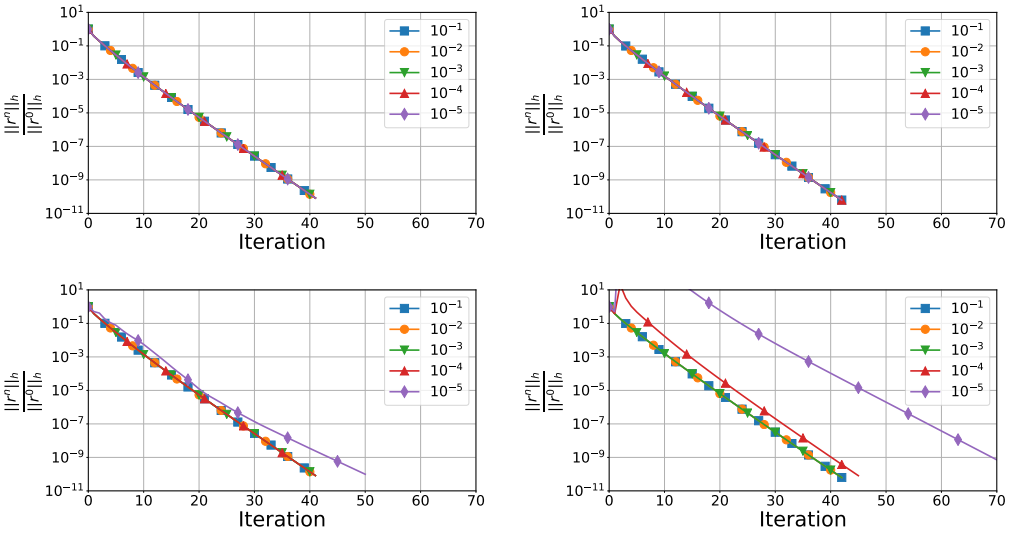

Figure 7.13: Number of iterations until convergence is reached for a selection of setups using BoxMG intergrid transfers, with checkerboard material parameter as the size of the jump increases. We compare two damped solvers, adAFAC-PI (left) vs. adAFAC-Jac (right). The top row shows the conventional method of precomputing all operators whereas the bottom row shows our delayed stencil integration with vertical rippling.

- Lazy integration + Geometric transfers: An incremental, lazy update of the fine grid stencils is computed in parallel with the multigrid solve, and an initial geometric guess for all coarse operators is used. Coarse grid updates ripple through the multigrid hierarchy via geometric intergrid transfer operators.

- Exact integration + BoxMG transfers: An exact computation of all fine grid is computed and all coarse grid operators using algebraic intergrid transfer operators (BoxMG) are setup prior to the first multigrid cycle.

- Lazy integration + BoxMG transfers: An incremental, lazy update of the fine grid stencils is computed in parallel with the multigrid solve, and an initial geometric guess for all coarse operators is used. Coarse grid updates ripple through the multigrid hierarchy via algebraic intergrid transfer operators (BoxMG).

- Lazy integration + Delayed BoxMG transfers: An incremental, lazy update of

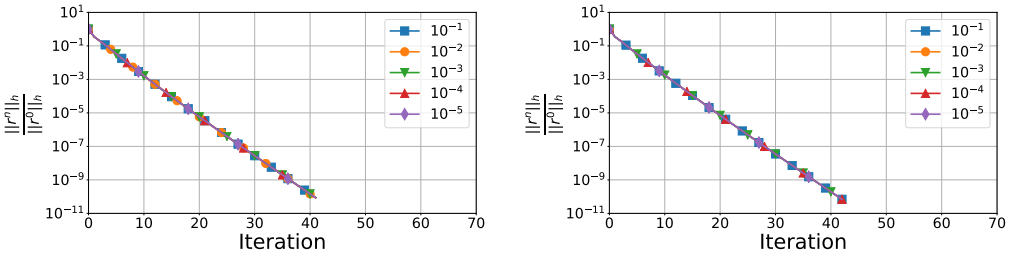

Figure 7.14: Number of iterations until convergence is reached for a selection of setups using BoxMG intergrid transfers, with checkerboard material parameter as the size of the jump increases. We show our delayed stencil integration with vertical rippling, but no coarse grid operator is used for an initial guess. We compare two damped solvers, adAFAC-PI (left) vs. adAFAC-Jac (right).

the fine grid stencils is computed in parallel with the multigrid solve, and no initial guess for the coarse operators is used. Instead coarse grid updates are disabled until the first algebraic expression (using algebraic grid transfers) has rippled through. Coarse grid updates ripple through the multigrid hierarchy via algebraic intergrid transfer operators (BoxMG).

In the final case, the algorithm develops from a 2-grid, into a 3-grid, into a 4-grid algorithm and so forth. We have split geometric and algebraic transfers into entirely separate setups due to the impact this choice has on vertical rippling.

When using geometric grid transfers, there are no sudden changes in coarse grid stencils between iterations. Ritz-Galerkin coarse grid operators with geometric grid transfers on regular grids produce the same stencils as rediscretisation with appropriately scaled basis functions. Recomputing coarse grid operators based on the currently held (intermediate) fine grid stencils is equivalent to computing rediscretised coarse grid operators with the same subcell material accuracy. This statement does not hold when algebraic grid transfers are used.

Our first tests focus on geometric grid transfers and vary k (Fig. 7.12). The damped additive solver variants outperform its plain additive cousin as plain additive multigrid either stagnates and the curve flattens off ($k = 1$) or becomes unstable and the

curve blows up ($k \geq 2$). The use of lazy evaluation does not impact performance, however. adAFAC-Jac is stable for all jump sizes and shows textbook multigrid convergence consistently. Additive multigrid and our damped additive solver show identical residual progression with both methods of stencil evaluation.

This does not remain the case when we use BoxMG algebraic grid transfers (Fig. 7.13). Two variants of our damped additive solver, adAFAC-PI and adAFAC-Jac with partially smoothed auxiliary restriction, show expected multigrid convergence for all k values when we precompute all operators. Our modified additive solvers only exhibit constant residual reduction as long as $k \leq 3$. If the parameter jump becomes bigger, they start to suffer from some instabilities in the first few iterations. This offsets the convergence curve. It is due to the rippling. Once we modify the rippling, i.e. involve only levels with reasonably valid operators and use no initial guess, we retain the convergence of accurate precomputation (Fig. 7.14).

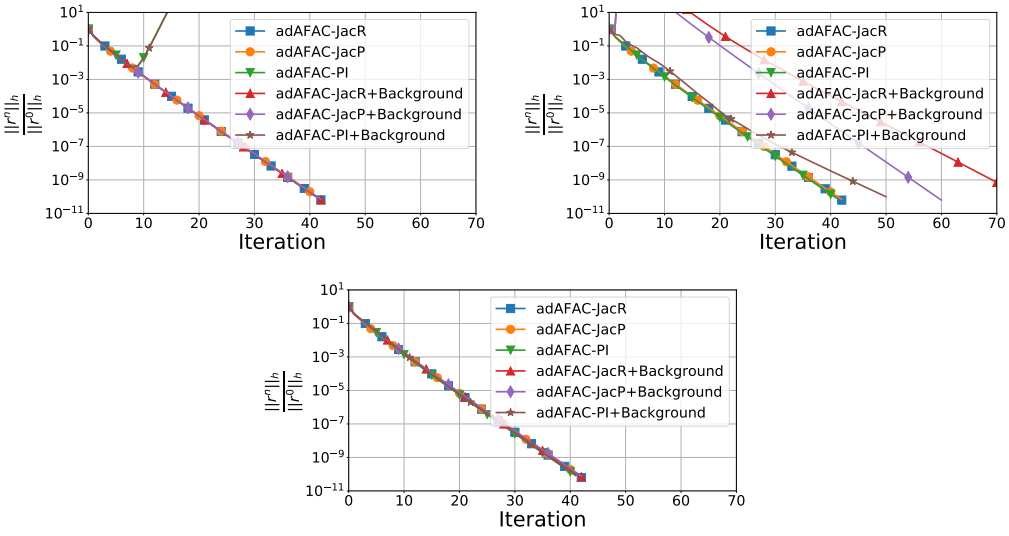

Figure 7.15: Number of iterations until convergence for $k = 5$. Top Left: We use geometric coarse operators and transfer operators. Top Right: We use BoxMG intergrid transfer operators and start from a geometric operator guess that we iteratively improve. Bottom: We use BoxMG intergrid transfer operators with no coarse grid operator initial guess.

We now fix $k = 5$ and further analyse the impact of the grid transfers on our damped

additive solvers. Our earlier experiments 7.3.3, suggest that additive multigrid which employs a solver that tests permanently on an auxiliary grid whether it overshoots is superior to a more "BPX-like" code, that is one which does not employ a coarse grid operator but solely uses grid transfers of fine grid corrections and residuals. With geometric transfers, there is again no noticeable impact from our lazy evaluation, even for this large k that can introduce instabilities in a solver (Fig. 7.15). Both variants of adAFAC-Jac converge and even though it starts to blowup, the residual development for adAFAC-PI remains similar when using lazy evaluation. adAFAC-Jac outperforms adAFAC-PI. With rippling, the coin flips. Now, these simpler solvers become superior as they do not require Ritz-Galerkin operators. The early shown offset for algebraic operators is bigger for the additive solvers that explicitly construct an additional coarse grid problem (adAFAC-Jac). We can eliminate this offset by negating the impact of algebraic coarse grids on the fine grids until an intermediate value has rippled to that level of the hierarchy. Multiplicative multigrid exhibits similarities to the resulting scheme. It first solves on the finest grid, then on the next coarser, then on the next coarser. In our additive mindset, the individual multiplicative steps are replaced by an additive cycle. However, if we omit this gradual switching on of coarser and coarser meshes, it seems that too wrong coarse grid operators can lead the solver into the wrong direction.

7.6.2 Rippling with dynamically adaptive meshes

We continue with experiments where the grid is no longer fixed. This adds an additional level of complexity, as the coarse grid operators used change through the solve, both due to the delayed fine grid integration plus the information rippling. In a traditional AMR/multigrid setup, any change in the grid necessitates a change in all "coarser" equations. This introduces a recompute step per mesh refinement. Our methodology hides the recomputation cost behind the solve. Unfortunately, information propagates at most one level per cycle up within the resolution hierarchy.

It is not clear whether such a massive delay in the coarse grid assembly could lead into stability problems or severe convergence penalties: The coarse grids are no longer acting upon the same equation as the fine grids all the time. While this is an effect affecting the previous experiments, due to the nature of additive multigrid, dynamic adaptive mesh refinement also makes the semantics of assembly matrices change: After each refinement, former fine grid discretisations suddenly become Ritz-Galerkin correction operators. With our tests, we investigate whether they still continue to push the solution in a direction that effectively minimises the error. We posit the improved stability from adAFAC might be of benefit. Our setup initially starts as a regular Cartesian mesh hosting 64 degrees of freedom on the fine grid and a single multigrid correction level. This increases due to the refinement to the order of 250,000 degrees of freedom and seven multigrid correction levels.

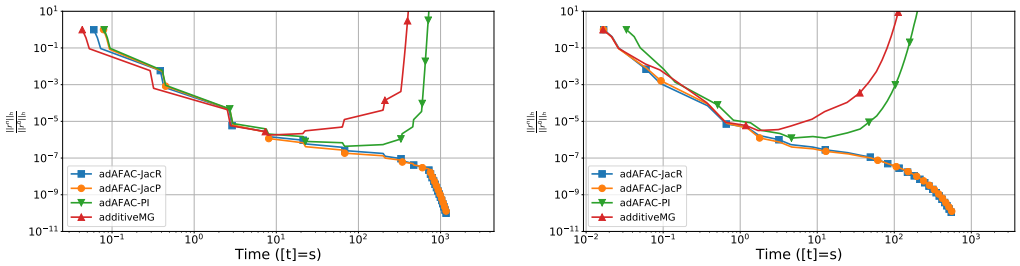

Figure 7.16: Residual plots for the jumping coefficient problem and $\epsilon \in \{10^{-3}, 1\}$ using geometric intergrid transfers. Both setups employ dynamically adaptive mesh refinement and either reassemble all operators accurately (left) or use delayed operator assembly (right).

We use our second test setup with only one type of a discontinuous material jump over three orders of magnitude. Initial results are given for $\epsilon \in \{10^{-3}, 1\}$. We further compare setups that use geometric grid transfer operators (Fig. 7.16) and algebraic grid transfer operators (Fig. 7.18). Our dynamic adaptivity criterion evaluates the solution's gradient over the domain after each multigrid cycle and picks the degrees of freedom carrying the top 10% of the absolute gradient values. We refine around these vertices and continue. Convergence requires a total number of updates in the order of 10^7 DoF updates. If a solve yields a residual that is 100 times bigger than

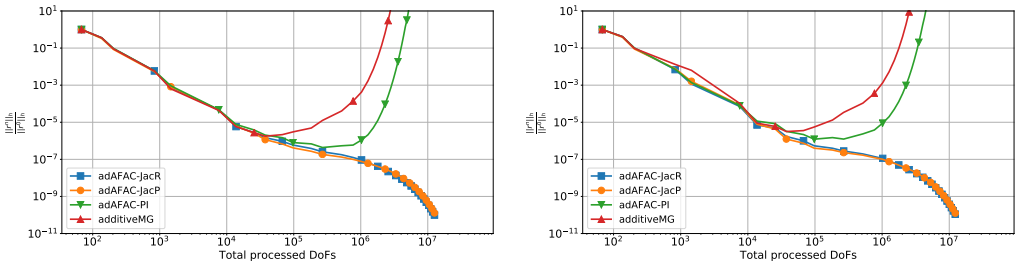

Figure 7.17: Residual plots for the jumping coefficient problem and $\epsilon \in \{10^{-3}, 1\}$ using geometric intergrid transfers. Both setups employ dynamically adaptive mesh refinement and either reassemble all operators accurately (left) or use delayed operator assembly (right). This is the same setup as in (Fig. 7.16) but comparing a normalised measure of computational cost rather than time directly.

the initial residual, we terminate the solver—even though the well-defined ellipticity implies that the solver eventually will "converge back". We perform two sets of comparisons: Firstly, we compare geometric intergrid transfers when all stencils are accurately computed before the solve resumes and when our lazy integratation is used. We compare this using the actual runtime and "Total Processed DoFs". We use this second metric—a count of the number of times a vertex is updated—so we can compare algorithmic development while taking into account the changing underlying mesh between iterations. Secondly, we compare algebraic intergrid transfers with lazy integration with the two possible methods of initialising coarse grid equations (either with an initial guess for the coarse or without any coarse grid impact at all).

For runs that converge, the code with a complete re-assembly after each refinement step converges with a rather shallow gradient at first. Throughout this phase, the grid is refined on alternate cycles. Once the grid becomes stationary, the solver exhibits a linear residual descent with a steeper gradient. We initially focus on geometric grid transfers with a fixed $k = 3$ (Fig. 7.16). At first glance, the two algorithms seem to progress quite differently, however this is not the case. If we compare the two setups relative to solution updates, as in (Fig. 7.17), we see that the residual progressions are similar. Comparing runtime directly takes into account

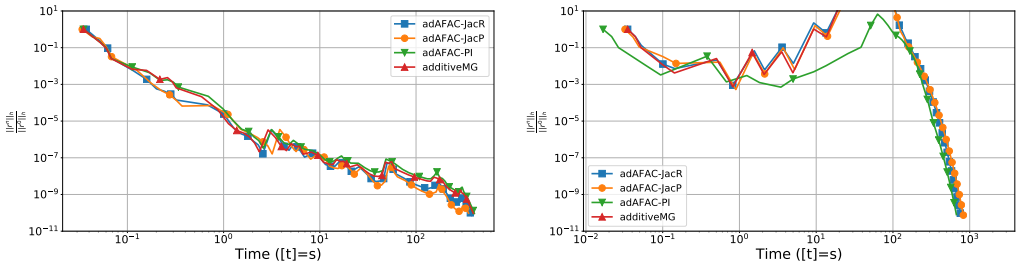

Figure 7.18: Residual plots for the jumping coefficient problem and $\epsilon \in \{10^{-3}, 1\}$ using BoxMG intergrid transfers. We show two setups that employ dynamically adaptive mesh refinement and a form of delayed operator assembly. After a refinement coarse grid operators either negate the impact of coarse levels temporarily (left) or the existing operators as initial guess (right).

the reassembly stages, which are not explicitly performed for our background stencil computation. Furthermore, we see the performance gains by the quicker time-to-solution. For such a large jump, the only setups that converge are damped additive setups that construct an explicit equation on the auxiliary grid, i.e. adAFAC-Jac variants. Visually, there is negligible difference in the plots for geometric intergrid transfer setups when precomputing all stencils and using our lazy evaluation. If all operators are algebraic (Fig. 7.18), both variants of adAFAC-Jac suffers from significant, temporary residual explosions which eventually are recovered when the coarse grid equations use an approximate initial guess. adAFAC-PI also exhibits this increase but not to quite as severe a degree. The undamped solver also exhibits this residual increases and is unable to recover. adAFAC-Jac is more robust yet still not as fast as its cousin with geometric intergrid transfer operators. We can massively reduce the residual increases by turning off coarse grids after a refinement and only re-enabling them once they have been set by a rippling process.

For both solver variants, our adaptive mesh refinement reduces the approximation accuracy temporarily, as it replaces mesh cells likely fed by high accuracy stencils with finer mesh cells with only one integration point. This induces oscillations manifesting in temporary residual spikes. As the fine grid cells start to improve their integration accuracy iteratively, the overall system accuracy recovers. Until this is

complete, the residual can continue to increase by many orders of magnitude, as the coarse grids solve an equation that is no longer a valid correction and hence push the solution into the wrong direction. With algebraic intergrid transfer operators, this effect is more distinct than with geometric operators: We know that geometric operators spanning big discontinuities induce oscillations on the fine grid. In the present case, we run into situations where algebraic intergrid transfer operators yield fine grid corrections anticipating the real material parameter behaviour, but the new fine grid discretisation is not yet ready to mirror them.

Observation 8. *Rippling can cause dynamic mesh refinement to introduce massive residual deterioration.*

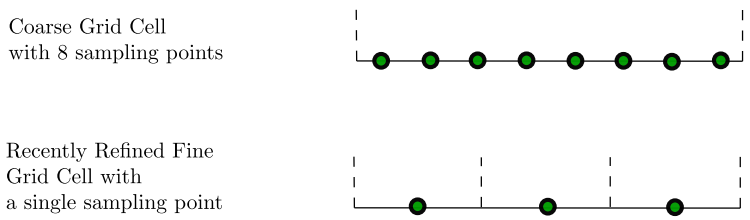

Figure 7.19: Illustration of material parameter sampling points in a coarse and fine stencil after a refinement. A reasonably accurate coarse stencil will use more sampling points than a recently instantiated fine grid stencil.

Rippling yields temporarily incompatible equation system configurations (Fig. 7.19). A newly introduced fine grid stencil will incorporate a less accurate material parameter than the existing coarse grid stencil. A straightforward fix to this behaviour would be a mechanism that ensures consistency of the total number and placement of material parameter sampling points in a region between the coarse grid level and new fine grid level: If a cell results from a coarse grid cell with n approximation points, it could immediately start a first fine grid integration with $\frac{n}{k^d}$ approximation points. However, such an approach would be antithetical to the idea behind the delayed approach. It would introduce new expensive assembly phases throughout the duration of the solve. Instead we borrow ideas from an FMG cycle—we do not incorporate corrections from all grid levels in one multigrid cycle. However,

we add additional coarse levels, rather than additional fine. We negate correction steps on coarse levels and only start using their impact in later iterations. The recently refined fine grid level ℓ is only impacted by coarse level $\ell - i$ after i multigrid cycles. The coarse grid operations ripple up the hierarchy one level per multigrid cycle. After each refinement, we therefore observe that the next finest level operator we depend on has been updated at least once before we compute a correction a coarse grid—either due to the Ritz-Galerkin recomputation or due to the delayed integration. This idea is repeatedly applied if we refine again and introduce an even finer grid level.

We recover the stability of multigrid with an explicit reassembly, although we need around twice as many DoF updates compared to a classical version. The improved stability is not dissimilar to classic multigrid theory where the fast F-cycle convergence rates require a higher order interpolation. We use classic d-linear interpolation here whenever we introduce new vertices. As we switch off the coarse grid corrections, we effectively smooth out this interpolation with a Jacobi step before we continue with multigrid. The multigrid in turn is not switched on immediately but we effectively work our way through a two-grid code, three-grid code, and so forth. In the first solver phase where we add new grid elements frequently, we only run series of fine grid smoothing steps for the majority of the cycles. The residual decays nevertheless, as most errors that can be resolved by newly introduced vertices here are high frequency errors which are damped out efficiently. At the same time, switching off coarse grid corrections tends to free compute resources which can be used to handle further stencil integrations.

Observation 9. *It is reasonable to pair up delayed stencil integration with a careful choice of which coarse grid operators are ready to be used in a multigrid cycle.*

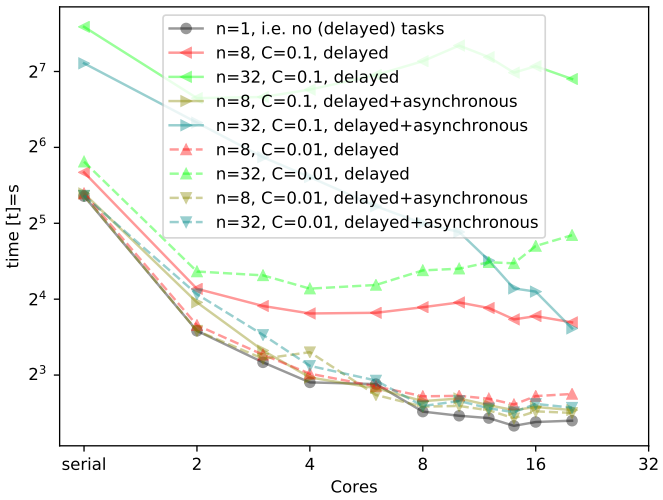

Figure 7.20: Run-time per grid sweep for twenty iterations for one discretisation with various integration/tasking configurations. Results for grid with $h \leq 0.005$. This corresponds to 58564 degrees of freedom.

7.7 Delayed stencil integration: Performance

We wrap up our experiments with simple single node studies—the tasking paradigm has sole single node effect. The experiments run through a series of setups per tested grid. We perform an equal splitting of elements per core, that is, each core handles the same number of degrees of freedom. First, we assess the pure scalability of the code without any delayed integration and furthermore fix the number of integration points $n = 1$. Next, we prescribe $n > 1$ and make the code yield $C \in \{0.1, 0.01\}$ integration tasks per cell, i.e. between one and ten percent of the cells spawn tasks. As pointed out before, this fraction in real applications is not fixed. We fix it manually here to assess the impact our idea has on scalability. Finally, we run each of the experiment with a delayed integration twice: In the baseline, the synchronisation is a preamble to the cell evaluation. In the alternative test, there is no synchronisation, i.e. we spawn the integration and do not wait for the result actively at any point. We work totally asynchronously. We test an arbitrary material parameter for the stencil integration as we are only interested in the workload and not the end result.

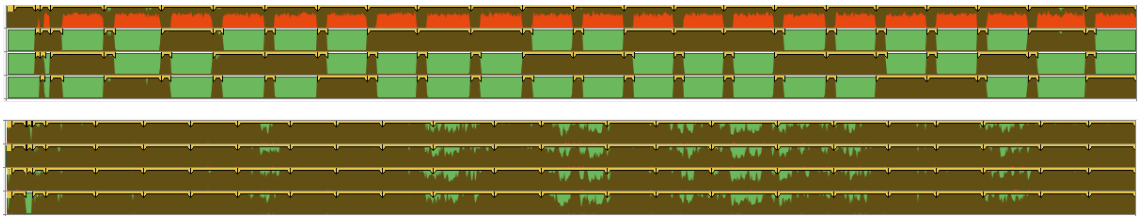

Figure 7.21: Task distribution/placement for one setup with four cores. Top: No delayed tasking. Bottom: Delayed and asynchronous tasking. Brown labels denote compute work, red is spinning (active waits), green denotes idling.

The partitioning with $n = 1$ yields reasonable performance (Fig. 7.20). This obviously is a "flawed" setup from a mathematical point of view, yet assesses that the underlying solver in principle does scale. As the workload is deterministic—it is hard-coded and does not use any additional tasking—the setup also clarifies that any tasking with $n > 1$ has to yield an unbalanced workload.

With integration for a ratio C of the cells, we indeed observe a deteriorated scalability. We can neglect any specific material parameter as we prescribe a fixed integration percentage—for all of our explored setups the high workload cells cluster along strong discontinuities. We use a geometric decomposition of the domain before we deploy the grid to the cores, and this decomposition tries to avoid disconnected partitions. As a consequence, one or few cores only are responsible for all the high-workload cells. With the anarchic tasking, we see that the scalability curve flattens out again and that we gain performance. This difference is greater with higher workload per integration and with higher core count.

Using Intel's VTune, we compare a setup that does not use delayed stencil integration (instead each immediately determines an improved operator during the grid traversal) and one that uses an anarchic, i.e. an asynchronous delayed operator integration (Fig. 7.21). We highlight a snapshot of a multigrid cycle for the two options, the total execution of the first option spans 709.5s (a priori integration) while the second covers 428.3s (asynchronous, delayed integration). This is a notable reduction in run-time per iteration. When anarchic tasking is not used, there are many periods where cores

are idle (shown in green) and only a single core is performing significant work (shown in brown). Although we have increased the core count we cannot achieve perfect scalability. From the colouring pattern, and frequent blocks of green, it is clear that cores idle and wait on a single core to finish its work before they themselves can perform meaningful work. A single core delays other cores as the stencil integration work is performed in sequence. Once the stencil integration is done, which core is idling switches and another core starts its work. By removing synchronisation points—and using anarchic tasking—when a core is idle it can instead pull the integration tasks from the task queue, rather than delaying other cores to compute the integration work. There are no more extended idling periods.

Observation 10. *The asynchronous, delayed element integration helps to regain some scalability for unbalanced setups.*

We observe that the cores that run out of work towards the end of their mesh traversal pick up some of the pending integration tasks spawned by overbooked colleague threads. Heavy integration tasks automatically slot into "idle" time of the baseline. The delayed, asynchronous integration yields a solver with a performance and scalability profile that is comparable to purely geometric multigrid where all operators are computed geometrically with $n = 1$ sampling points per cell. Our scalability tests fix the fraction C of cells that require an improved integration as well as n. They thus study only the scalability behaviour of one particular multigrid cycle. If we study the whole time-to-solution of a solver, we find that this behaviour typically translates into a walltime of around 2/3 of the baseline (Fig. 7.4). Baseline here is an implementation that uses the exactly same code base, yet realises the lazy evaluation pattern, i.e. computes all operators prior to the first usage accurately. Walltime always comprises both assembly phase and solve phase.

7.8 Wrap-up

We have shown that adAFAC-x, in both its adAFAC-PI and adAFAC-Jac forms, is an effective class of solver and is consistent with additive multigrid. It produces consistent solutions for the default Poisson equation, but it does not exhibit the same overshooting additive multigrid shows once the problem size grows. Furthermore, it remains stable once we introduce challenging material coefficients that cause vanilla additive multigrid to diverge. Oscillations are effectively damped out. Our target implementation is generic and takes advantage of mature techniques—such as BoxMG intergrid transfer operators and dynamically adaptive grids through HTMG. We have seen effective scaling and performance for our adAFAC-x implementations. The additional workload from auxiliary grids does not significantly effect time taken per multigrid cycle or limit the parallel scalability.

Lazily constructing fine grid equations also produces consistent solutions to constructing them a priori. Multigrid using asynchronous stencil assembly is an actual solver. It is not only valid as a preconditioner. Stencils are assembled non-deterministically in parallel but we have not observed issues wherein a solver converges to an incorrect solution due to inaccurate locally held stencils. If a solver converged using a priori stencil integration, then it still converges when switching assembly method. When using geometric intergrid transfers, the delayed fine grid equation construction does not introduce any additional instabilities within a solve. Algebraic intergrid transfers require additional care so convergence rates are not harmed but can still remain effective. On average, we have observed a reduction in time-to-solution of around 2/3 due to our use of asynchronous assembly and we have also seen a significant improvement in single node core utilisation when using parallel tasks—previously idle periods in the multigrid cycle have been eliminated. Single node scaling is also promising.

Chapter 8

Conclusion

In this thesis, we have presented two new ideas that push multigrid further along the path to exascale. These are our delayed integration of the integrations for the fine grid matrix and our adAFAC-x solver suite. Although neither idea are a silver bullet that reinvents multigrid, they are both salves for lycanthropy that introduce additional potential for parallelism in multigrid. Both of our novel ideas are implemented within a single-touch context on an adaptive space-tree. Multigrid is notoriously hard to mesh with other solvers due to the requirements when constructing the grid hierarchy. Our implementation is shown to be compatible with mature techniques, such as HTMG and BoxMG. As we are single-touch, the implementation inherits good cache locality.

8.1 Discussion of our findings

8.1.1 Asynchronous assembly

Our delayed stencil integration pipelines the assembly process so that it now overlaps with early multigrid cycles. We reorder computations in the assembly phase and break them down into a series of iterative tasks that can be performed in parallel with early iterations of the solve. Instead of using the true discretisation in the earlier

multigrid iterations, the solver uses an initial approximation. For most situations, this inaccurate approximation has been shown to not harm the rate of convergence. The assembly phase is known to have lower arithmetic intensity than other phases, therefore our reordering of operations eliminates algorithmic latency and increases concurrency earlier in the solve. This is the reason we achieve an overall reduction in time-to-solution even though we have increased the total assembly workload.

These two features—the increase in possible concurrency and corresponding reduction in algorithmic latency—are the main gains seen from asynchronous assembly. Multigrid cycles start earlier in the run-time. We have shown that most of the assembly workload can be deployed to cores within a machine that would otherwise by idle within a multigrid iteration. There are two extreme cases in the assembly process where our solver is ill-suited: When there is a negligible integration setup (i.e. constant material parameter, so we perform redundant work with an iterative integration); and when the integration is especially challenging (i.e. it requires a long sequence of our iterative stencil integrations before we obtain an accurate stencil). In the former case we have not empirically seen a notable performance penalty. The latter case might cause a "starvation effect" and new integrations may not drop in, or the frequent stencil changes might introduce instabilities. These instabilities are due to coarse grid levels aggressively correcting and causing temporary overshooting. Our tasking methodology has exhibited no issues due to starvation—the solver never terminated early, giving an incorrect solution due to the true fine grid equation not being available. We did see some setups requiring additional solver iterations to converge—but the solver always converged to the true solution. While starvation is still a theoretical concern, we do not believe it to be an issue in the main.

We have shown that, for a multigrid solver that uses geometric grids and rediscretisation, the impact on convergence due to our asynchronous assembly is negligible. Geometric definitions create a hierarchy of equations, where all coarse grid equations can be approximated with relatively high accuracy using only geometric information. More algebraically inspired setups, such as those that use operator induced restric-

tion or prolongation, require additional information from the fine grids to construct the coarse grids. This obviously increases the cost of assembly when switching from geometric to algebraic setups and introduces additional dependencies on the coarse grid levels. Both could possibly limit the effectiveness of an asynchronous assembly process. We have shown that with careful handling of coarse grids, our asynchronous methodology can still be effective in the presence of algebraic grid transfers. Early solver iterations use a modified grid cycle, inspired by F-cycles, so that coarse grid equations are setup in tandem with coarse grid levels impacting fine grid levels for the first time.

Lazy assembly has also been demonstrated to be particularly well-suited for dynamically adaptive meshes. A dynamically adaptive mesh already amortises the cost of assembly throughout the solve but historically has required many re-assembly phases to accommodate this. We hide these re-assembly phases within the solve itself. The solver does not pause after a refinement to wait for updated fine grid equations to be computed—instead, solver iterations continue unabated after a refinement and the asynchronous assembly computes the updated fine grid stencils in the background. Switching to asynchronous assembly reduces time-to-solution.

8.1.2 Damping term

We have also shown that our family of adAFAC-x damping parameters is an effective method of stabilising additive multigrid. Additive multigrids exhibits instabilities due to both the total number of grid levels increasing, and the introduction of complicated domain features, such as discontinuous material parameters. We have introduced a per-vertex damping parameter that reduces overcorrections between levels, thereby improving stability. This improvement has empirically been shown to be similar the improvement when switching from geometric to algebraic intergrid transfer operators (e.g. BoxMG). adAFAC-Jac with geometric transfer operators is approximately as stable as additive multigrid with BoxMG transfer operators.

Moreover, the damping parameter does not significantly impact the overall rate of convergence of the solve when additive multigrid is stable as we compute the damping parameter using an additional coarse grid solve on an auxiliary grid. We augment intergrid transfer operators for the auxiliary grid, i.e. we smooth them, to anticipate coarse grid smoothing. The auxiliary grid solve is computed totally asynchronously to other corrections, therefore it does not impact the parallelism within additive multigrid. The smoothing is also cheap, so it does not introduce a significant additional workload. The construction of the damping term is not tied into any one implementation, therefore using the damping terms does not conflict with techniques such as BoxMG, both can be used in tandem to further improve stability. Although we have implemented our damping methodology using single-touch ideas on space-trees, the construction of damping term could also meld with any target mesh or underlying additive multigrid solver.

adAFAC-x auxiliary corrections are cheap to compute. The construction of the auxiliary meshes and equations does not require any additional data structures or setup time—it can repurpose existing grids and equations. Most multigrid implementations are memory bound—so the increase in computational cost of another mat-vec per iteration does not impact run-time significantly. The required matrix is already loaded into memory per iteration and applying the matrix to another vector per iteration is not an expensive procedure. Auxiliary grid data transfers also maintain existing data locality of the solver, as they mirror existing intergrid transfers. There is, however, an additional memory cost of storing the smoothed intergrid transfers. The stencils for the smoothed intergrid transfer can be hard-coded if the fine grid equation is homogeneous, but once the material parameter varies, this becomes infeasible so they must instead be computed at run-time and stored. Overall, computing our auxiliary damping term has a reasonably low impact on the total computational cost of each iteration of the solve itself.

An obvious point of comparison for our damping parameter is the AFACx scheme and its damping parameter designed to remove “oscillatory components”. Both schemes

are additive and compute damping terms using auxiliary grids. The additive nature means the damping parameters are computed independently of the coarse correction and applied additively. Therefore both share a similar operation count and core idea. However, AFACx produces the fine correction sequentially after the auxiliary correction. We do not share this limitation—both corrections can be computed in parallel. Therefore, from an implementational perspective we have an advantage.

8.2 Future work

8.2.1 Addressing weaknesses and shortcomings

Although we have used HTMG, and therefore a FAS based implementation, we have not directly dealt with nonlinear equations. We have used the principles, but only to simplify the handling of AMR—we have not implemented a true nonlinear solver. Both our additive damping ideas and asynchronous/lazy method of assembly are applicable here. In a nonlinear solver, the local operator is dependent upon the solution at that point, therefore the operator applied in each iteration is not fixed throughout the solve as the locally held solution is updated. Changing the operator bears similarity to how we update the representation of the fine grid equation throughout the solve. Both operators are merely approximations of the true operator—which is only available at the end of the solve. Nonlinear equations are more challenging to solve than linear equations and might cause solvers to exhibit instabilities. Therefore improving stability, via our damping procedure, would also benefit the solve.

The impact of different smoothers on both our additive damping parameter and delayed stencil integration is a change that would notably alter their behaviour. Currently, we have only used simple Jacobi smoothers. We could investigate the impact that more powerful smoothers—such as block smoothers—could have on our additive damping parameters specifically the intergrid transfer operators for the

auxiliary grids. Block smoothers would involve increasing the support of the intergrid transfers. We could also crib an idea from mult-additive—which uses a specific symmetrised smoother on the coarse grid to more closely represent a $V(1,1)$ cycle while still being totally additive. Using this smoother on the auxiliary grid would allow us to also be closer to a $V(1,1)$ cycle—rather than a $V(1,0)$ or $V(0,1)$ cycle. Block smoothers are again an interesting prospect for delayed stencil integration. We would deploy both the block smoothing and stencil assembly to a background task and the smoothing and assembly would be performed therein. Assembling a stencil/smoother for block smoothing is more expensive than assembling a Jacobi smoother, so is a good fit for our lazy approach.

8.2.2 Integrating our ideas within other solvers

Currently we have only developed a standalone multigrid solver. This demonstrates that our ideas work, but is of limited use in a wider context, or to a wider audience. Our key ideas work best as a single tool within a wider toolkit rather than as the only tool within a shed. Developing a solver/module that uses our ideas and integrates within another system, such as an elliptic submodule within PETSc, is a natural progression. However, due to our geometric coarsening, our additive solver may only mesh well with a limited subset of existing solver hierarchies. We would need to explore other coarsening procedures or methods of choosing initial/tentative intergrid transfer operators. A common practice in the multigrid community is to fuse multiple multigrid solvers into the same multigrid cycle. They use one multigrid flavour on the fine levels and a different flavour on the coarse. This can be seen in many implementations, such as *hp*-multigrid [123], any flavour of algebraic-geometric multigrid [73], [124], and many existing large scale runs that already already couple additive solvers in a multigrid hierarchy [93], [94]. Our damped additive scheme could fit into such a solver stack. Again, our choice of geometric coarsening may create challenges. It is not trivial to construct a geometric multigrid scheme on top of an algebraic scheme. In addition, if an existing solver cannot represent

the smoother as a sparse matrix product, then smoothing the intergrid transfer operators for our auxiliary grids may also prove to be non-trivial: They may result in intergrid transfers which are dense, which destroys data locality and the simplicity that adAFAC-x takes advantage of. We have also not fully explored the impact of coupling a fine grid scheme that uses high order polynomials with coarse grids that are lower order—such aggressive coarsening is known to worsen convergence rates. As our solver is readily parallel we would remove potential choke points on coarser grid levels. In a stack of solvers, with each solver having its own construction phase, assembly costs can be a concern. Using our asynchronous assembly to accelerate part of, or all of, the stack is therefore an avenue of interest. Coarser solvers here act as approximate solvers to accelerate the solve—they are not exact. Our approximate assembly should not hinder convergence. However, as discussed in Section 4.9, there may be implementational issues when using asynchronous assembly on less simple smoothers.

The work on "Asynchronous Multigrid" of Chow and Wolfson-Hou has already been discussed in Section 4.10, where we compared their notion of asynchronous multigrid to ours. They modify multigrid to eliminate synchronisation between smoothing steps in additive multigrid, whereas we modifying multigrid to eliminate synchronisation between the assembly phase and the solver phase. Both approaches to asynchronicity are orthogonal to each other and could be used in concert. Their asynchronous multigrid method uses different threads—that do not synchronise updates—to smooth different regions. We could simply instantiate these threads with approximations of the true smoother and over the course of the solve feed these threads with updated approximations of the true matrix equation.

8.2.3 Application to new problems

We have only applied our solver to a specific example of an elliptic equation. It would be equally well suited to a wider array of problems, for example, we can

apply both our lazy stencil integration and damping parameter to other elliptic equations—not just to Poisson's equation. An obvious candidate for our lazy stencil integration is more complicated operators with costly stencil integration. When assembly is more expensive there are greater gains to be made by pipelining the assembly—a larger workload can potentially be parallelised. Our current work uses relatively simple elliptic equations, locally regular grids and simple nodal basis functions. Basis functions that are d-linear are cheap to integrate and the structure of the mesh also further reduces the integration cost. All these factors mean we may not currently see the full potential of an asynchronous assembly process. Instead applying a delayed stencil integration procedure to a fine grid that uses higher order basis functions, bubble functions, or any equation that requires a more expensive numerical integration, increases the possible concurrency earlier in the solve for that solver.

On the other hand, we could move in the other direction and focus on further minimising the cost of assembly. This follows on from the work on compressed stencil storage to produce a quasi-matrix-free setup [33]. We have already performed preliminary experiments in this vein, where our asynchronous assembly takes advantage of approximated stencil storage and mixed precision computations [3]. Our current work approximates stencils using low order integrations and with appropriate handling, we see minimal deterioration in convergence rates. Further approximating stencils using lower accuracy floating pointing storage would reduce the memory footprint and arithmetic latency when fetching stencils from memory. This could yield further reductions in time-to-solution.

A powerful additive solver, such as adAFAC-x, can be applied to other setups that are more challenging. Incompressible Navier-Stokes often represents the pressure component as a Laplacian that must be solved once for each time step solved. Here the pressure component is likely to only change by a small amount between time steps—so a single iteration of a multigrid solver will be sufficient to update the pressure between time steps. Our solvers single-touch policy and embedding

into a single mesh traversal is therefore well suited. Alternatively, rather than applying our solver as a subsystem it could be directly applied to a parabolic system, solving a convection dominated problem as the sole solver, or as a multigrid-in-time solver—building upon the work of Weinzierl and Köppl [125]—and work directly on space-time grids.

Bibliography

[1] C. D. Murray and T. Weinzierl, 'Stabilized asynchronous fast adaptive composite multigrid using additive damping', *Numerical Linear Algebra with Applications*, 2020, doi:https://doi.org/10.1002/nla.2328.

[2] ——, 'Lazy stencil integration in multigrid algorithms', in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2019, pp. 25–37, doi:https://doi.org/10.1007/978-3-030-43229-4_3.

[3] ——, 'Delayed approximate matrix assembly in multigrid with dynamic precisions', *Concurrency and Computation: Practice and Experience*, 2020, doi:https://doi.org/10.1002/cpe.5941.

[4] W. Hackbusch, *Elliptic differential equations: theory and numerical treatment*. Springer, 2017, vol. 18.

[5] T. W. Secomb, R. Hsu, E. Y. Park and M. W. Dewhirst, 'Green's function methods for analysis of oxygen delivery to tissue by microvascular networks', *Annals of Biomedical Engineering*, vol. 32, no. 11, pp. 1519–1529, 2004.

[6] E. Hernández-Baltazar and J. Gracia-Fadrique, 'Elliptic solution to the Young–Laplace differential equation', *Journal of Colloid and Interface Science*, vol. 287, no. 1, pp. 213–216, 2005.

[7] M. Tuller, D. Or and D. Hillel, 'Retention of water in soil and the soil water characteristic curve', *Encyclopedia of Soils in the Environment*, vol. 4, pp. 278–289, 2004.

[8] A. J. Chorin, J. E. Marsden and J. E. Marsden, *A mathematical introduction to fluid mechanics*. Springer, 1990, vol. 168.

[9] A. J. Chorin, 'A numerical method for solving incompressible viscous flow problems', *Journal of Computational Physics*, vol. 135, no. 2, pp. 118–125, 1997.

[10] J.-L. Guermond and L. Quartapelle, 'On stability and convergence of projection methods based on pressure Poisson equation', *International Journal for Numerical Methods in Fluids*, vol. 26, no. 9, pp. 1039–1053, 1998.

[11] J.-L. Guermond and A. Salgado, 'A splitting method for incompressible flows with variable density based on a pressure Poisson equation', *Journal of Computational Physics*, vol. 228, no. 8, pp. 2834–2846, 2009.

[12] W. L. Briggs, V. E. Henson and S. F. McCormick, *A multigrid tutorial*. SIAM, 2000.

[13] U. Trottenberg, C. W. Oosterlee and A. Schüller, *Multigrid*. Academic Press, 2001.

[14] P. Bastian, G. Wittum and W. Hackbusch, 'Additive and multiplicative multi-grid a comparison', *Computing*, vol. 60, no. 4, pp. 345–364, 1998.

[15] L. Hart and S. F. McCormick, 'Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Basic ideas', *Parallel Computing*, vol. 12, pp. 131–144, 1989.

[16] S. F. McCormick and D. J. Quinlan, 'Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Performance results', *Parallel Computing*, vol. 12, no. 2, pp. 145–156, 1989.

[17] S. F. McCormick, *Multilevel projection methods for partial differential equations*. SIAM, 1992.

[18] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku and B. Braunschweig, 'The international exascale software project roadmap', *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[19] A. Brandt, 'Multi-level adaptive techniques (MLAT) for singular-perturbation problems', *Numerical Analysis of Singular Perturbation Problems*, pp. 53–142, 1979.

[20] ——, 'Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems', in *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, Springer, 1973, pp. 82–89.

[21] J. E. Dendy, 'Black box multigrid', *Journal of Computational Physics*, vol. 48, no. 3, pp. 366–386, 1982.

[22] T. Grauschopf, M. Griebel and H. Regler, 'Additive multilevel preconditioners based on bilinear interpolation, matrix-dependent geometric coarsening and algebraic multigrid coarsening for second-order elliptic PDEs', *Applied Numerical Mathematics*, vol. 23, no. 1, pp. 63–95, 1997.

[23] W. Hackbusch, *Theory and numerical treatment of elliptic differential equations*. Springer, Berlin, 1992.

[24] ——, *Iterative solution of large sparse systems of equations*. Springer, 1994, vol. 95.

[25] J. H. Ferziger, M. Perić and R. L. Street, *Computational methods for fluid dynamics*. Springer, 2002, vol. 3.

[26] H. Ji, F.-S. Lien and E. Yee, 'An efficient second-order accurate cut-cell method for solving the variable coefficient Poisson equation with jump conditions on irregular domains', *International Journal for Numerical Methods in Fluids*, vol. 52, no. 7, pp. 723–748, 2006.

[27] M. Ainsworth and C. Glusa, 'Aspects of an adaptive finite element method for the fractional Laplacian: A priori and a posteriori error estimates, efficient implementation and multigrid solver', *Computer Methods in Applied Mechanics and Engineering*, vol. 327, pp. 4–35, 2017.

[28] M. Paszyński, D. Pardo, C. Torres-Verdín, L. Demkowicz and V. Calo, 'A parallel direct solver for the self-adaptive hp finite element method', *Journal of Parallel and Distributed Computing*, vol. 70, no. 3, pp. 270–281, 2010.

[29] M. Dumbser and R. Loubère, 'A simple robust and accurate a posteriori sub-cell finite volume limiter for the discontinuous Galerkin method on unstructured meshes', *Journal of Computational Physics*, vol. 319, pp. 163–199, 2016.

[30] A. Düster, J. Parvizian, Z. Yang and E. Rank, 'The finite cell method for three-dimensional problems of solid mechanics', *Computer Methods in Applied Mechanics and Engineering*, vol. 197, no. 45-48, pp. 3768–3782, 2008.

[31] J. Parvizian, A. Düster and E. Rank, 'Finite cell method', *Computational Mechanics*, vol. 41, no. 1, pp. 121–133, 2007.

[32] B. Reps and T. Weinzierl, 'A complex additive geometric multigrid solver for the Helmholtz equations on spacetrees', *ACM Transactions on Mathematical Software*, vol. 44, no. 1, 2:1–2:36, 2017.

[33] M. Weinzierl and T. Weinzierl, 'Quasi-matrix-free hybrid multigrid on dynamically adaptive Cartesian grids', *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, pp. 1–44, 2018.

[34] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster and S. Wild, 'Applied mathematics research for exascale computing', DOE ASCR Exascale Mathematics Working Group, 2014.

[35] R. S. Sampath and G. Biros, 'A parallel geometric multigrid method for finite elements on octree meshes', *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1361–1392, 2010.

[36] M. J. Berger and J. Oliger, 'Adaptive mesh refinement for hyperbolic partial differential equations', *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[37] M. J. Berger and P. Colella, 'Local adaptive mesh refinement for shock hydrodynamics', *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[38] L. Hart, S. F. McCormick, A. O'Gallagher and J. Thomas, 'The fast adaptive composite-grid method (FAC): Algorithms for advanced computers', *Applied Mathematics and Computation*, vol. 19, no. 1-4, pp. 103–125, 1986.

[39] M. Griebel and G. Zumbusch, 'Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization', *Contemporary Mathematics*, vol. 218, pp. 271–278, 1998.

[40] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk and G. Biros, 'Dendro: Parallel algorithms for multigrid and AMR methods on 2: 1 balanced octrees', in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, p. 18.

[41] H. Sundar, R. S. Sampath and G. Biros, 'Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel', *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2008.

[42] T. Weinzierl and M. Mehl, 'Peano – A traversal and storage scheme for octree-like adaptive Cartesian multiscale grids', *SIAM Journal on Scientific Computing*, Special Section: 2010 Copper Mountain Conference, vol. 33, no. 5, R. Tuminaro, M. Benzi, X.-C. Cai *et al.*, Eds., pp. 2732–2760, 2011.

[43] C. Burstedde, L. C. Wilcox and O. Ghattas, 'P4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees', *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[44] T. Weinzierl, 'The Peano software—parallel, automaton-based, dynamically adaptive grid traversals', *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 2, p. 14, 2019.

[45] A. Brandt, 'Guide to multigrid development', in *Multigrid Methods*, Springer, 1982, pp. 220–312.

[46] J. W. Ruge and K. Stüben, 'Algebraic multigrid', in *Multigrid methods*, SIAM, 1987, pp. 73–130.

[47] J. Xu and L. Zikatanov, 'Algebraic multigrid methods', *Acta Numerica*, vol. 26, pp. 591–721, 2017.

[48] K. Stüben, 'An introduction to algebraic multigrid', *Multigrid*, pp. 413–532, 2001.

[49] K. Stüben, 'A review of algebraic multigrid', *Numerical Analysis: Historical Developments in the 20th Century*, pp. 331–359, 2001.

[50] S. Maclachlan and N. Madden, 'Robust solution of singularly perturbed problems using multigrid methods; analysis and numerical results in one and two dimensions', Technical report, Department of Mathematics, Tufts University, Tech. Rep., 2012.

[51] C.-T. Wu and H. C. Elman, 'Analysis and comparison of geometric and algebraic multigrid for convection-diffusion equations', *SIAM Journal on Scientific Computing*, vol. 28, no. 6, pp. 2208–2228, 2006.

[52] F. O. Campos, R. S. Oliveira and R. W. dos Santos, 'Performance comparison of parallel geometric and algebraic multigrid preconditioners for the bidomain equations', in *International Conference on Computational Science*, Springer, 2006, pp. 76–83.

[53] B. Amaziane, A. Bourgeat and J. Koebbe, 'Numerical simulation and homogenization of two-phase flow in heterogeneous porous media', in *Mathematical Modeling for Flow and Transport Through Porous Media*, Springer, 1991, pp. 519–547.

[54] J.-F. Bourgat, 'Numerical experiments of the homogenization method', in *Computing Methods in Applied Sciences and Engineering, 1977, I*, Springer, 1979, pp. 330–356.

[55] S. Knapek, 'Matrix-dependent multigrid homogenization for diffusion problems', *SIAM Journal on Scientific Computing*, vol. 20, no. 2, pp. 515–533, 1998.

[56] M. G. Edwards and C. F. Rogers, 'Multigrid and renormalization for reservoir simulation', in *Multigrid Methods IV*, Springer, 1994, pp. 189–200.

[57] P. R. King, 'The use of renormalization for calculating effective permeability', *Transport in Porous Media*, vol. 4, no. 1, pp. 37–58, 1989.

[58] N. Neuss, W. Jäger and G. Wittum, 'Homogenization and multigrid', *Computing*, vol. 66, no. 1, pp. 1–26, 2001.

[59] S. Wang and E. Sturler, 'Multilevel sparse approximate inverse preconditioners for adaptive mesh refinement', *Linear Algebra and its Applications*, vol. 431, no. 3-4, pp. 409–426, 2009.

[60] U. Rüde, *Mathematical and computational techniques for multilevel adaptive methods*. SIAM, 1993.

[61] R. E. Alcouffe, A. Brandt, J. E. Dendy Jr and J. W. Painter, 'The multi-grid method for the diffusion equation with strongly discontinuous coefficients', *SIAM Journal on Scientific and Statistical Computing*, vol. 2, no. 4, pp. 430–454, 1981.

[62] J. D. Moulton, J. E. Dendy Jr and J. M. Hyman, 'The black box multigrid numerical homogenization algorithm', *Journal of Computational Physics*, vol. 142, no. 1, pp. 80–108, 1998.

[63] A. Greenbaum, 'Analysis of a multigrid method as an iterative technique for solving linear systems', *SIAM Journal on Numerical Analysis*, vol. 21, no. 3, pp. 473–485, 1984.

[64] ——, 'A multigrid method for multiprocessors', *Applied Mathematics and Computation*, vol. 19, no. 1-4, pp. 75–88, 1986.

[65] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro and U. M. Yang, 'A survey of parallelization techniques for multigrid solvers', in *Parallel Processing for Scientific Computing*, SIAM, 2006, pp. 179–201.

[66] J. E. Jones and S. F. McCormick, 'Parallel multigrid methods', in *Parallel Numerical Algorithms*, Springer, 1997, pp. 203–224.

[67] J. H. Bramble, J. E. Pasciak and J. Xu, 'Parallel multilevel preconditioners', *Mathematics of Computation*, vol. 55, no. 191, pp. 1–22, 1990.

[68] T. Chan and R. Tuminaro, 'Design and implementation of parallel multigrid algorithms', in *In Proceedings Third Copper Mountain Conference on Multigrid Methods*, SIAM, 1987, pp. 101–115.

[69] R. S. Tuminaro, 'A highly parallel multigrid-like method for the solution of the euler equations', *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 1, pp. 88–100, 1992.

[70] J. E. Dendy and J. D. Moulton, 'Black box multigrid with coarsening by a factor of three', *Numerical Linear Algebra with Applications*, vol. 17, pp. 577–598, 2010.

[71] I. Yavneh and M. Weinzierl, 'Nonsymmetric black box multigrid with coarsening by three', *Numerical Linear Algebra with Applications*, vol. 19, no. 2, pp. 246–262, 2012.

[72] P. M. De Zeeuw, 'Matrix-dependent prolongations and restrictions in a blackbox multigrid solver', *Journal of Computational and Applied Mathematics*, vol. 33, no. 1, pp. 1–27, 1990.

[73] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas and G. Stadler, 'Parallel geometric-algebraic multigrid on unstructured forests of octrees', in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, 2012, p. 43.

[74] S. P. MacLachlan, J. D. Moulton and T. P. Chartier, 'Robust and adaptive multigrid methods: Comparing structured and algebraic approaches', *Numerical Linear Algebra with Applications*, vol. 19, no. 2, pp. 389–413, 2012.

[75] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick and J. Ruge, 'Adaptive smoothed aggregation (α sa) multigrid', *SIAM Review*, vol. 47, no. 2, pp. 317–346, 2005.

[76] ——, 'Adaptive algebraic multigrid', *SIAM Journal on Scientific Computing*, vol. 27, no. 4, pp. 1261–1286, 2006.

[77] S. S. Vangara, A. Kashi and S. Nadarajah, 'Additive multigrid with scaled correction for implicit compressible flow solvers', in *AIAA Aviation 2019 Forum*, 2019, p. 3712.

[78] B. Smith, P. Bjorstad and W. Gropp, *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, 2004.

[79] X. Zhang, 'Multilevel schwarz methods', *Numerische Mathematik*, vol. 63, no. 1, pp. 521–539, 1992.

[80] X.-C. Tai and P. Tseng, 'Convergence rate analysis of an asynchronous space decomposition method for convex minimization', *Mathematics of Computation*, vol. 71, no. 239, pp. 1105–1135, 2002.

[81] B. Lee, S. F. McCormick, B. Philipp and D. J. Quinlan, 'Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations', *SIAM Journal Numerical Analysis*, vol. 42, pp. 130–152, 2004.

[82] D. J. Quinlan, 'Adaptive mesh refinement for distributed parallel architectures', PhD thesis, University of Colorado at Denver, Foo, Jul. 1993.

[83] A. Brandt, 'Multi-level adaptive techniques (MLAT) for partial differential equations: Ideas and software', in *Mathematical Software*, Elsevier, 1977, pp. 277–318.

[84] ——, 'Multi-level adaptive solutions to boundary-value problems', *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977.

[85] S. F. McCormick and J. Thomas, 'The fast adaptive composite grid (FAC) method for elliptic equations', *Mathematics of Computation*, vol. 46, no. 174, pp. 439–456, 1986.

[86] P. K. Jimack and M. A. Walkley, 'Asynchronous parallel solvers for linear systems arising in computational engineering', *Computational Technology Reviews*, vol. 3, pp. 1–20, 2011.

[87] H.-J. Bungartz and M. Griebel, 'Sparse grids', *Acta Numerica*, vol. 13, pp. 147–269, 2004.

[88] O. A. McBryan, 'Parallel superconvergent multigrid', *Multigrid Methods: Theory, Applications, and Supercomputing*, vol. 110, p. 195, 1988.

[89] P. S. Vassilevski and U. M. Yang, 'Reducing communication in algebraic multigrid using additive variants', *Numerical Linear Algebra with Applications*, vol. 21, no. 2, pp. 275–296, 2014.

[90] J. Wolfson-Pou and E. Chow, 'Asynchronous multigrid methods', in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2019, pp. 101–110.

[91] B. Gmeiner, H. Köstler, M. Stürmer and U. Rüde, 'Parallel multigrid on hierarchical hybrid grids: A performance study on current high performance computing clusters', *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 217–240, 2014.

[92] P. T. Lin, J. N. Shadid, J. J. Hu, R. P. Pawlowski and E. C. Cyr, 'Performance of fully-coupled algebraic multigrid preconditioners for large-scale vms resistive mhd', *Journal of Computational and Applied Mathematics*, vol. 344, pp. 782–793, 2018.

[93] D. A. May, J. Brown and L. Le Pourhiet, 'A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous stokes flow', *Computer Methods in Applied Mechanics and Engineering*, vol. 290, pp. 496–523, 2015.

[94] J. Rudi, A. C. I. Malossi, T. Isaac, G. Stadler, M. Gurnis, P. W. Staar, Y. Ineichen, C. Bekas, A. Curioni and O. Ghattas, 'An extreme-scale implicit solver for complex PDEs: Highly heterogeneous flow in earth's mantle', in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, ACM, 2015, pp. 1–12.

[95] M. Griebel, *Zur lösung von finite-differenzen-und finite-element-gleichungen mittels der hierarchischen-transformations-mehrgitter-methode [On the solution of the finite-difference and finite-element equations through the hierarchical-transformational-multigrid method]*. Technische Universität München. Institut für Informatik, 1990.

[96] P. Ghysels, T. J. Ashby, K. Meerbergen and W. Vanroose, 'Hiding global communication latency in the GMRES algorithm on massively parallel

machines', *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. C48–C71, 2013.

[97] P. Ghysels and W. Vanroose, 'Hiding global synchronization latency in the preconditioned conjugate gradient algorithm', *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014.

[98] P. Ghysels, P. Kłosiewicz and W. Vanroose, 'Improving the arithmetic intensity of multigrid with the help of polynomial smoothers', *Numerical Linear Algebra with Applications*, vol. 19, no. 2, pp. 253–267, 2012.

[99] P. Ghysels and W. Vanroose, 'Modeling the performance of geometric multigrid stencils on multicore computer architectures', *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C194–C216, 2015.

[100] M. Mehl, T. Weinzierl and C. Zenger, 'A cache-oblivious self-adaptive full multigrid method', *Numerical Linear Algebra with Applications*, vol. 13, no. 2–3, pp. 275–291, 2006.

[101] F. Günther, M. Mehl, M. Pögl and C. Zenger, 'A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves', *SIAM Journal on Scientific Computing*, vol. 28, no. 5, pp. 1634–1650, 2006.

[102] D. Braess, *Finite elements: Theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, 2007.

[103] D. E. Knuth, 'The genesis of attribute grammars', in *WAGA: Proceedings of the International Conference on Attribute Grammars and their Applications*, P. Deransart and M. Jourdan, Eds., Paris, France: Springer-Verlag, 1990, pp. 1–12.

[104] R. S. Tuminaro and C. Tong, 'Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines', in *Supercomputing, ACM/IEEE 2000 Conference*, IEEE, 2000, pp. 5–5.

[105] P. Vaněk, J. Mandel and M. Brezina, 'Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems', *Computing*, vol. 56, no. 3, pp. 179–196, 1996.

[106] P. Vaněk, 'Fast multigrid solver', *Applications of Mathematics*, vol. 40, no. 1, pp. 1–20, 1995.

[107] J. Bjørgen and J. Leenaarts, 'Numerical non-LTE 3D radiative transfer using a multigrid method', *Astronomy & Astrophysics*, vol. 599, A118, 2017.

[108] J. Kouatchou and J. Zhang, 'Optimal injection operator and high order schemes for multigrid solution of 3D Poisson equation', *International Journal of Computer Mathematics*, vol. 76, no. 2, pp. 173–190, 2000.

[109] W. H. Press and S. A. Teukolsky, 'Multigrid methods for boundary value problems. i.', *Computers in Physics*, vol. 5, no. 5, pp. 514–519, 1991.

[110] A. Brandt, 'General highly accurate algebraic coarsening', *Electronic Transactions on Numerical Analysis*, vol. 10, no. 1, p. 21, 2000.

[111] R. D. Falgout, 'An introduction to algebraic multigrid', *IEEE Annals of the History of Computing*, vol. 8, no. 06, pp. 24–33, 2006.

[112] R. Moe, *Iterative local uniform mesh refinement methods and parallel processing*. University of Bergen. Department of Informatics, 1992.

[113] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[114] A. Bienz, R. D. Falgout, W. Gropp, L. N. Olson and J. B. Schroder, 'Reducing parallel communication in algebraic multigrid through sparsification', *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S332–S357, 2016.

[115] R. D. Falgout and J. B. Schroder, 'Non-Galerkin coarse grids for algebraic multigrid', *SIAM Journal on Scientific Computing*, vol. 36, no. 3, pp. C309–C334, 2014.

[116] D. E. Charrier, B. Hazelwood and T. Weinzierl, 'Enclave tasking for DG methods on dynamically adaptive meshes', *SIAM Journal on Scientific Computing*, vol. 42, no. 3, pp. C69–C96, 2020.

[117] D. E. Charrier, B. Hazelwood, E. Tutlyaeva, M. Bader, M. Dumbser, A. Kudryavtsev, A. Moskovsky and T. Weinzierl, 'Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver', *The International Journal of High Performance Computing Applications*, vol. 33, no. 5, pp. 973–986, 2019.

[118] J. E. Dendy Jr, 'Black box multigrid for nonsymmetric problems', *Applied Mathematics and Computation*, vol. 13, no. 3-4, pp. 261–283, 1983.

[119] S. Balay, S. Abhyankar, M. F. Adams *et al.*, *PETSc web page*, `https://www.mcs.anl.gov/petsc`, 2021, `https://www.mcs.anl.gov/petsc`.

[120] ——, 'PETSc users manual', Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.15, 2021, `https://www.mcs.anl.gov/petsc`.

[121] S. Balay, W. D. Gropp, L. C. McInnes and B. F. Smith, 'Efficient management of parallelism in object oriented numerical software libraries', in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset and H. P. Langtangen, Eds., Birkhäuser Press, 1997, pp. 163–202.

[122] A. Gholami, D. Malhotra, H. Sundar and G. Biros, 'FFT, FMM, or multigrid? a comparative study of state-of-the-art Poisson solvers for uniform and nonuniform grids in the unit cube', *SIAM Journal of Scientific Computing*, vol. 38, no. 3, pp. C280–C306, 2016.

[123] C. R. Nastase and D. J. Mavriplis, 'High-order discontinuous Galerkin methods using an hp-multigrid approach', *Journal of Computational Physics*, vol. 213, no. 1, pp. 330–357, 2006.

[124] M. Weinzierl, 'Hybrid geometric-algebraic matrix-free multigrid on spacetrees', PhD thesis, Technische Universität München, 2013.

[125] T. Weinzierl and T. Köppl, 'A geometric space-time multigrid algorithm for the heat equation', *Numerical Mathematics: Theory, Methods and Applications*, vol. 5, no. 1, pp. 110–130, 2012.

[126] D. A. Beard and J. B. Bassingthwaighte, 'Modeling advection and diffusion of oxygen in complex vascular networks', *Annals of biomedical engineering*, vol. 29, no. 4, pp. 298–310, 2001.