Received: 19 February 2020 | Revised: 12 May 2020 | Accepted: 9 June 2020
DOI: 10.1002/cpe.5941

SPECIAL ISSUE PAPER

Delayed approximate matrix assembly in multigrid with dynamic precisions

Charles D. Murray | Tobias Weinzierl

Department of Computer Science, Durham University, Durham, UK

Correspondence
Charles D. Murray, Department of Computer Science, Durham University, Lower Montjoy, South Road, DH1 3LE Durham, UK.
Email: c.d.murray@durham.ac.uk

Funding information
Engineering and Physical Sciences Research Council, Grant/Award Number: 1764342; ExaHyPE, Grant/Award Number: 671698

Summary

The accurate assembly of the system matrix is an important step in any code that solves partial differential equations on a mesh. We either explicitly set up a matrix, or we work in a matrix-free environment where we have to be able to quickly return matrix entries upon demand. Either way, the construction can become costly due to nontrivial material parameters entering the equations, multigrid codes requiring cascades of matrices that depend upon each other, or dynamic adaptive mesh refinement that necessitates the recomputation of matrix entries or the whole equation system throughout the solve. We propose that these constructions can be performed concurrently with the multigrid cycles. Initial geometric matrices and low accuracy integrations kickstart the multigrid iterations, while improved assembly data is fed to the solver as and when it becomes available. The time to solution is improved as we eliminate an expensive preparation phase traditionally delaying the actual computation. We eliminate algorithmic latency. Furthermore, we desynchronize the assembly from the solution process. This anarchic increase in the concurrency level improves the scalability. Assembly routines are notoriously memory- and bandwidth-demanding. As we work with iteratively improving operator accuracies, we finally propose the use of a hierarchical, lossy compression scheme such that the memory footprint is brought down aggressively where the system matrix entries carry little information or are not yet available with high accuracy.

KEYWORDS
algebraic-geometric multigrid, asynchronous multigrid, delayed operator computation, dynamically adaptive Cartesian grids, finite element assembly, mixed precision computing

1 | INTRODUCTION

Multigrid algorithms are among the fastest solvers known for elliptic partial differential equations (PDEs) of the type

$$-\nabla(\epsilon\nabla)u = f \tag{1}$$

on a d-dimensional, well-shaped domain Ω. An approximation of the function $u : \Omega \mapsto \mathbb{R}$ is what we are searching for with $\epsilon : \Omega \mapsto \mathbb{R}^+$ as a material parameter, and $f : \Omega \mapsto \mathbb{R}$ constituting the right-hand side. The system is closed by appropriate boundary conditions. We restrict ourselves to

This paper is an extended version of C.D. Murray and T. Weinzierl: Lazy stencil integration in multigrid algorithms as introduced and published at the PPAM'19 Conference.

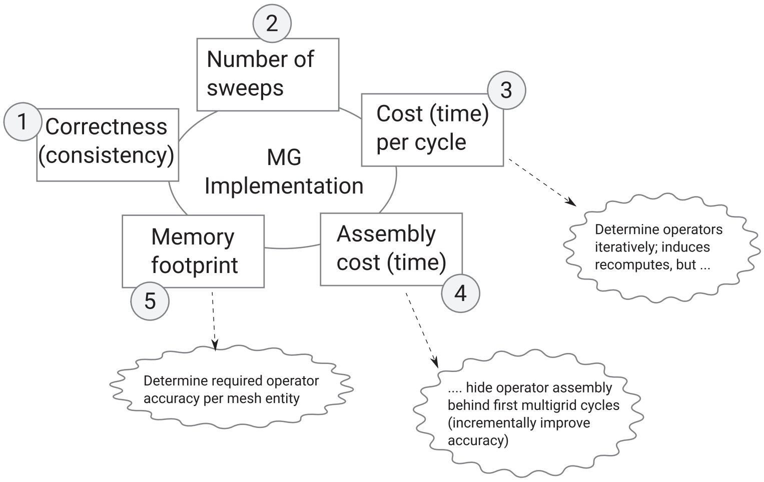

1 Correctness (consistency)
2 Number of sweeps
3 Cost (time) per cycle
MG Implementation
Memory footprint 5
Assembly cost (time) 4
Determine operators iteratively; induces recomputes, but ...
Determine required operator accuracy per mesh entity
.... hide operator assembly behind first multigrid cycles (incrementally improve accuracy)

FIGURE 1 Multigrid implementation challenges: (1) An implementation has to be correct, that is, yield the result of the underlying mathematics and thus be consistent with it, (2) there has to be minimal data structures, while the cost per cycle (3), the assembly/setup cost (4) and the memory footprint (5) have to be small, too. (3),(4),(5) are the core areas where we make a contribution, while implications on (1) and (2) are studied

Dirichlet conditions here. A Ritz-Galerkin finite element discretization over a mesh Ω_h that geometrically discretizes Ω yields an equation system $A_h u_h = f_h$. This is the linear equation system actually tackled by multigrid (MG). Equations of the type (1) arise in many application domains studying (quasi-)stationary phenomena. They also are an important building block within many time-dependent problems, where they model incompressibility conditions or friction for example. Finally, they also arise in Lagrangian setups, where they model gravity between moving objects for example.

Common to all sketched application areas is that solving $A_h u_h = f_h$, that is, finding $u_h \approx A_h^{-1} f_h$, is expensive. The ellipticity of Equation (1) implies that any correction of the solution anywhere within Ω impacts the solution over the entirety of Ω. Finite elements and related techniques manage to break up this strong global dependency by discretizing the PDE with test and shape functions that have local support. Any single update of an entry of u_h affects only neighboring elements within the discretization, that is, few other entries within u_h. It propagates through the whole domain from there. We work with equation systems that are sparse, and thus manageable from both a memory and compute effort point of view. However, information propagation this way is intrinsically slow. Multigrid compensates for this effect as it removes errors across a hierarchy of meshes. Coarser and coarser grids take ownership of updates that yield nonlocal modifications, that is, they handle low-frequency errors from the fine grid. This multilevel approach to tackle an elliptic problem on a cascade of resolutions is the seminal idea behind multigrid and the reason why multigrid is fast.

When we implement multigrid, the implementation has to be in line with the mathematical theory, while the cost of the actual implementation is determined by at least four factors: the number of solver iterations; the time taken for a single solver iteration including all data traversals—there can be multiple of them in the MG context; the compute cost (time) to set up all required operators (matrices) on all of the different scales and all required operators coupling different scales; and the memory footprint of the method (Figure 1). Traditionally, computer science research focuses on the first two aspects—both from a numerical point of view and with performance engineering glasses on. Yet, there are more and more cases where the latter two constraints become prescient. There are at least three reasons for this. First, the proportion of the runtime spent on the solution phase often diminishes relative to (re-)assembly time: If Equation (1) serves as a building block within a time-stepping code, a solution u will often not change radically between time steps. Along the same lines, dynamic adaptivity changes the mesh and consequently demands for a new discretization of Equation (1). Yet, a grid update usually does not change the solution completely. As a result, a low number of iterations or cycles yield a valid solution in both cases as long as we start from the previous step's solution as an initial guess. Second, we suffer from a widening memory gap on our machines.[1] The increase in memory access speed cannot keep pace with the growth of compute power on newer systems. As dynamic adaptivity—and nonlinear systems which are out of scope here—become mainstream, we assemble multiple times. Conversely one assembly remains sufficient for static meshes and linear PDEs. Each reassembly stresses the memory interconnects since it is typically not compute-intense. Yet, this is the step where volumetric domain information (ϵ-fields) has to be streamed into the core. The impact is amplified by multigrid's inherent multiscale nature: It is not only a single matrix A_h discretized Equation (1) rather we have to maintain and construct a series of matrices $A_h, A_{kh}, A_{k^2h}, \ldots$ for coarsening by a factor of $k \geq 2$, as well as the corresponding inter-grid transfer operators, that is, prolongations and restrictions. Finally, due to this multitude of involved operators, multigrid is memory-demanding. In an era with stagnating or even decreasing memory per core, the overhead to store restriction, prolongation and coarse grid matrices quickly becomes a limiting constraint. Purely geometric approaches with rediscretization avoid this overhead. They construct local matrix entries on-demand, usually when they are required throughout a matrix-vector product. In such a case, there is no need to store the matrix entries at all. It is, however, known that massive sudden ϵ changes, additional convective terms, or nonlinearities here require very detailed operator recomputations. We trade the memory demands for heavy compute effort. Furthermore, geometric multigrid tends to become unstable in case of nontrivial ϵ distributions[2-4] and is not stable in a multigrid sense either.[5]

The additional operators introduced by multigrid on top of the actual discretization are correction operators, while the overall scheme is an iterative approach. Furthermore, the fundamental solutions to Equation (1) quickly decay. Ergo, neither do the coarse grid equations have to tackle the exact equation right from the start, nor do we need exact fine grid operators prior to the first iteration. Furthermore, if an operator is slightly wrong, that is, if it yields slightly wrong iterates, this error will mainly affect the area around the erroneous update in subsequent applications, as the fundamental solutions to Equation (1) quickly decay. Therefore, it is sufficient to kick off with approximate operators, as long as (i) we later increase their accuracy such that we eventually solve the right equation system; (ii) the smoother continually pushes the solution into the right direction; (iii) the correction equations do not impede these improvements; and (iv) we quickly get the majority of operators within the computational domain right. Our idea is, therefore, to kick off multigrid with very crude fine grid approximations plus geometric coarse grid and inter-grid transfer operators which can be quickly precomputed. While we run the multigrid cycle, we successively improve the approximation quality of the fine grid operators. These improvements are deployed as tasks which run in the background of the actual solver: While our solver determines initial solution approximations, we derive the correct operators describing the true solution. The overall integration is precision-guided, that is, we continuously improve those equation system entries that continue to benefit from improved integration as well as improved storage accuracy. There is neither a global, uniform numerical integration precision nor a uniform global (IEEE) floating point format. The whole mindset unfolds its beauty once we stop storing multigrid's matrices explicitly. Instead, we embed the matrix entries into our mesh—a strategy we label as quasi matrix-free[5]—and store solely differences to geometric operators. The information within any low accuracy stencil, no matter whether sufficient or not yet available with higher accuracy, can then be encoded with few bytes only. Overall, our contribution is three-fold:

1. We allow for a solver start without an expensive pre-solve assembly phase, that is, we eliminate algorithmic latency;
2. We increase the code's concurrency level as we decouple the actual assembly process from the solve and make it feed into the latter anarchically, that is, as soon as results become available yet without any synchronization or temporal ordering;
3. And we bring down the memory footprint. This notably affects the initial steps which tend to be cheap anyway, as dynamic grid refinement just starts to unroll the real compute grid.

To the best of our knowledge, this is the first algorithmic blueprint systematically exploring how to incorporate problem-dependent numerical integration of fine grid stencils, adaptive coarse grid operator computation, and non-IEEE storage formats without the cost of any arduous assembly into one multigrid implementation.

The remainder of this article is organized as follows: We first discuss potential caveats of our philosophy in Section 2 that we want to keep in mind and address throughout the article. We then review published and related work, which contextualizes the present work, explains where ideas come from and how the proposed techniques fit to other activities. All details required to understand the novel ideas of our work are introduced together with our multigrid algorithm of interest in Section 4. From there, we establish our notion of a delayed, iterative stencil assembly with flexible precision (Section 5). We dedicate Section 6 to a discussion of the solver's properties that result from this novel assembly paradigm. A brief summary and an outlook close the discussion.

2 | SHORTCOMINGS OF THE PROPOSED CONCEPTS

There are certain obvious caveats with our proposed solution:

1. The matrix entry computation could be subject to starvation. If the results of new integration tasks are not dropping in on time, the multigrid solver might converge towards the wrong solution and prematurely signal termination. If the integration is run with high priority and thus precedes the actual solution process, it is not subject to these concerns. Yet, it loses the proposed selling point. One might argue that complex integration patterns typically arise only around submanifolds within the domain. Despite our remarks on a rapid decay of errors, the elliptic nature of the problem of choice, however, implies that inaccurate integration within a subdomain can pollute the entire solution. The starvation phenomenon thus has to be analyzed. We have to validate that the total number of cycles/iterations required is not increased massively (item (2); Figure 1).
2. The motivation behind multigrid's construction—we work with a cascade of coarser and coarser grids where the coarse grids "take care" of global propagation—implies that any fine grid stencil modification has its impact felt throughout all grid levels. As long we change the discretization, the exact nature of all coarse grid operators also continues to change. To avoid repeated data accesses per sweep, we limit the multilevel propagation speed, that is, our operator updates propagate from fine to coarse one level per cycle. Updates ripple through the system. For multiplicative multigrid, such a one-level-at-a-time policy is reasonable, as the solver also handles one solution at a time. However, additive multigrid processes all resolutions in one rush. Coarse operators thus are incorrect if operator changes are not immediately rolled out to all levels. Therefore, we exclusively focus on additive multigrid here. It is more challenging. The rippling then has to be anticipated carefully—in particular once we work

with dynamically adaptive meshes—as we run into risks of coarse grid updates pushing the solution in the wrong direction, that is, would yield inconsistent update rules (item (1); Figure 1). Nonlinear setups would yield the same issues.

3. Our techniques are not lightweight in a sense that they can easily be added to existing solvers in a black-box fashion. This particularly holds for the usage of multiple precision formats, for which most software might be ill-prepared. While our experiments focus on structured adaptive meshes resulting from spacetrees only, it is clear that they apply directly to unstructured meshes even if they lack a built-in hierarchy, since both geometric and algebraic coarsening construct a hierarchy anyway. Our accuracy dependencies and recomputation need to follow this hierarchy. It is, however, clear that the integration of the ideas into existing unstructured mesh software requires additional effort and software design.

Our experiments suggest that the first item is not observed in practice, even though our work studies a worst-case setup with our additive MG solvers. We propose a solution to the second caveat. For the third item, our work provides clues for what solver development roadmaps might want to incorporate in the future.

3 RELATED WORK AND INFLUENCING IDEAS

Early work by Achi Brandt[6]—specifically his work on MLAT—already clarifies that "discretization and solution processes are intermixed with, and greatly benefit from, each other" in an adaptive scheme. This principle is not exclusive to MLAT. It holds for all forms of adaptive mesh refinement (AMR). If dynamic AMR starts with a coarse discretization and unfolds the mesh anticipating the real solution's behavior, we can read this as delaying an exact computation of the fine grid equations until we know that they are needed. Dynamic adaptivity also interleaves the assembly with the linear equation system solver.

Within the solver world, the seminal additive scheme proposed by Bramble et al [7,8] does not utilize exact equation representations on coarse grid levels. Multiple coarse grid corrections are computed independently from the same fine grid residual and eventually summed up. However, the coarse grid correction equation in a multigrid sense results from a simple *h*-scaling of the diagonal of the fine grid equation's operator. This approximation is sufficient for convergence in many cases.[9] Starting with inexact equations or setups and improving them subsequently is not new within the multigrid-as-a-solver community either. Adaptive AMG[10,11] constructs coarse grids iteratively. A tentative coarsening setup is improved by applying it to a set of candidate vectors for a small number of iterations. Similarly, Bootstrap AMG[12] modifies both prolongations and the coarse grid hierarchy itself by applying them to randomly constructed problems and modifying them to improve convergence. In both Adaptive AMG and Bootstrap AMG, all modifications of the coarse grid equations happen during an extended setup phase rather than during the iterations of the solution process.

Inaccurate operator approximations are popular when solving nonlinear equations. In the inexact newton method[13,14] for example, the Jacobian is approximated once and then used within an iterative process yielding a sequence of corrections to the solution. Along the same lines, multigrid for nonlinear problems often uses multigrid within a Newton solve where the nonlinear operator is linearized, that is, approximated. Finally, some multigrid techniques for convection-dominated scenarios symmetrize the underlying fine grid phenomenon[15] before they make it subject to algebraic coarsening. The motivation here is to stabilize the solver, but it obviously constructs a regime with inaccurate coarse grid operators. Our solver overview is far from comprehensive.

For time-dependent problems, many established (commercial) codes still employ direct solvers that store an explicit inverse of the system matrix. This is particularly attractive in scenarios where the mesh does not change, as the inversion of a matrix preceding a new time step's solution is performed only once. After that, we merely apply the known inverse, that is, rely on a matrix-vector product. The massive memory footprint required to store an inverse of a sparse system makes this approach quickly prohibitive. However, applying multigrid to each and every time step also yields excessive costs, as the solution changes smoothly in time. We therefore have previously studied a "multigrid" concept where a single level smoothing step is followed by a two-level scheme, followed by another single level solve, a four level scheme, and so forth.[16] The coarser a grid level, the more it affects future solutions even though we do not update the underlying operator in time anymore. Such a scheme translates the *h*-coarsening idea of multigrid into the time domain. Most approaches from the parallel-in-time community (see Reference 17 and follow-up work for newer trends) also exploit the idea to approximate the coarse operators crudely, but to incrementally improve them.

On the implementation side, a lot of mature software supports matrix-free solvers today. PETSc for example phrases its algorithms as if it had a fully assembled matrix at hand. However, many of its core routines make no assumptions about how this matrix is actually assembled. In its MatShell variant, it specifically allows users to deliver matrix parts on-demand,[18] that is, PETSc asks for the matrix part, applies it to the vectors of interest, and immediately discards the operator again. While such an approach allows users to (re-)compute all operators whenever they are required, a sole matrix-free implementation runs the risk to quickly become inefficient or unstable. If material parameters in Equation (1) change rapidly, any on-demand operator evaluation has to integrate the underlying weak formulation with high accuracy. This quickly becomes expensive. On the coarser grids, a sole matrix-free mindset means that operators cannot depend on the next finer operator following a Ritz-Galerkin multigrid strategy, as this next finer operator is not available explicitly either and will recursively depend on even finer levels. For these shortcomings, the hybridization of algebraic and geometric multigrid is well-trodden ground.[19-22] Most hybrid approaches use algebraic multigrid where sole geometric operators fail, but employ geometric operators wherever possible. As material parameters change "infrequently" on fine meshes and diffusion dominates in many setups, it is indeed possible to rely on a geometric operator construction for the majority of the equation systems.

Our own work in Reference 5 introduces an alternative notion of hybrid algebraic-geometric multigrid. All operators here are embedded into the grid and, in principle, algebraic. As they are encoded within the mesh, we can access them within a matrix-free mindset. Storing the operators relative to geometric operators in a compressed form allows us to work with a memory footprint close to geometric multigrid. This article follows-up on this strategy and fuses the underlying idea of lossy compression with iterative operator assembly. It stores operators with reduced precision where appropriate. We have previously explored lossy low precision storage both in a multigrid[5] and an SPH[23] context. Both approaches demonstrate how much we can save in terms of memory footprint, and both publications point out that the price to pay for this is an increased arithmetic workload. In the SPH context, we propose and prototype how the additional operators can be deployed to tasks of its own. However, the strict causal dependency there implies that performance penalties can and do arise. Performance penalties resulting from the increased compute load—it is notably more expensive to check to which degree we can store an operator with reduced precision—do not arise in the present case, as we propose an anarchic scheme where we ignore dependencies of the actual solve on the newly introduced stencil assembly tasks plus any storage loss analysis.

4 | A MATRIX-FREE ADDITIVE MULTIGRID SOLVER ON SPACETREES

Let $A_h u_h = f_h$ describe the equation system that arises from a nodal Ritz-Galerkin finite element discretization of Equation (1). The test and shape function space are the same. We use d-linear functions. Hence, we obtain the classic 9-point or 27-point stencils on regular meshes. Each stencil describes the entries of one row of A_h. It describes how a single point (vertex) on one level depends on its cell-connected neighbors. The solution vector u_h stores the weights (scalings) of the individual shape functions. This simple choice of mathematical ingredients allows for a multitude of multigrid flavors already. We classify some of these flavors and point out which flavors we study in this article. The list is not comprehensive but focuses on solver nuances which are affected by the proposed techniques.

Geometric vs. algebraic construction of multilevel hierarchy. Multigrid solvers can be classified into either mesh-based coarsening or solvers with algebraic coarsening.[24,25] The former rely on an existing cascade of coarser and coarser geometric meshes $\Omega_h, \Omega_{kh}, \Omega_{k^2h}, \dots$ which often embed into each other. Algebraic coarsening derives the coarse meshes from a connectivity analysis of A_h, that is, directly from the matrix without a geometric interpretation. In our work, we stick to a geometric approach relying on spacetrees[26]: We take the domain Ω and embed it into a cube. The cube yields a mesh Ω_0 without any real degrees of freedom, as all vertices either discretize points outside of the domain or coincide with the domain boundary. We cut the cube into k equidistant slices along each coordinate axis to end up with k^d new cubes which form Ω_1. On a cube-by-cube basis we continue recursively yet independently.

The construction process yields a tree of cubes which define a cascade of grids $\Omega_0, \Omega_1, \dots, \Omega_{\ell_{\max}}$ that are embedded into each other. We call the subscript ℓ in Ω_ℓ the grid level, that is, the smaller the index the coarser the mesh,[27,28] and make a few observations: The individual grids embed into coarser grids ($\Omega_{\ell-1} \subset \Omega_\ell$). Individual meshes Ω_ℓ might cover only parts of the domain and might yield disjoint submeshes. The union of all meshes $\underset{\ell}{\cup}\Omega_\ell = \Omega_h$ is an adaptive mesh. By subsequently removing the largest level ℓ from the union, we can construct our coarse grid hierarchy in a geometric multigrid sense. With this level definition, it is convenient to label u_ℓ with the level instead of a generic $_h$ subscript.

Correction vs. full approximation storage realizations. Multigrid solvers can be classified into correction schemes and full approximation storage (FAS) realizations.[28,29] The latter operate on a solution to the PDE on each and every level, whereas in a classic correction scheme, the coarse grid weights have solely correcting semantics. We make all inner and boundary points of each mesh Ω_ℓ carry a d-linear shape function. $\underset{\ell}{\cup}\Omega_\ell = \Omega_h$ hence spans a generating system where multiple vertices (and therefore weights) coincide spatially yet are unique due to their level. Let $P_\ell{}^{\ell+1}$ denote prolongations of data on level ℓ onto level $\ell+1$. As there is no guarantee that all cubes of the mesh Ω_ℓ are refined, the operator might only take a subset of Ω_ℓ and transfer it onto the next resolution level. Let $R_\ell{}^{\ell-1}$ be a restriction, that is, the counterpart operation to $P_\ell{}^{\ell+1}$. Again, it affects only those regions of Ω_ℓ that are refined further. There are three natural implications of this setup: (i) We can make the overall solution to the PDE unique by enforcing $u_{\ell-1}(x) \leftarrow u_\ell(x)$ for every vertex pair that coincides spatially. We write this down as $u_{\ell-1}(x) = I_\ell{}^{\ell-1} u_\ell(x)$. I is the injection operator[30] (also named "trivial restriction"[27]). (ii) We can simply interpolate weights from $u_{\ell-1}$ to all hanging vertices on level ℓ. Hanging vertices, that is, vertices with less than 2^d adjacent cells on the same level, do not carry any real shape functions. Yet, we temporarily augment them by truncated shapes such that a weak Ritz-Galerkin formulation for their neighboring nonhanging vertices on the same level makes sense. (iii) We can exploit the Ritz-Galerkin coarse grid operator definition and implement Griebel's HTMG[30] straightforwardly:

$$\underbrace{A_{\ell-1}}_{=R_\ell^{\ell-1}A_\ell P_{\ell-1}^\ell} u_{\ell-1} = A_{\ell-1}e_{\ell-1} + A_{\ell-1}I_\ell^{\ell-1}u_\ell = R_\ell^{\ell-1}\underbrace{r_\ell}_{=f_\ell - A_\ell u_\ell} + R_\ell^{\ell-1}A_\ell P_{\ell-1}^\ell I_\ell^{\ell-1}u_\ell = R_\ell^{\ell-1}\left(f_\ell - A_\ell \underbrace{\left(u_\ell - P_{\ell-1}^\ell I_\ell^{\ell-1}u_\ell\right)}_{\hat{u}_\ell}\right)$$

This is an elegant rephrasing of FAS relying on two different types of residuals: the standard residual $r_\ell = f_\ell - A_\ell u_\ell$ guides any iterative update on a level, while its hierarchical counterpart $\hat{r}_\ell = f_\ell - A_\ell \hat{u}_\ell$ feeds into its next coarser equation. Both result from the same discretization stencil. FAS traditionally is introduced as a tool to handle nonlinear equation systems. It also pays off for in-situ visualization as it holds the solution in a multiscale representation and thus can allow users to zoom in and out. It encodes levels of detail. We use FAS as it makes the handling of hanging meshes simple[2,5]: Fine grid vertices adjacent to a refinement transitions on the finest level have two different semantics: They carry correction and solution weights. Both quantities work in different regimes—the solution weights converging towards the "real" value while the corrections approach zero. With FAS, we do not have to distinguish/classify them. This argument rolls over recursively to all grid levels.

Rediscretization and geometric projections between levels vs. Ritz-Galerkin plus algebraic operators. Multigrid solvers can be classified into solvers using rediscretization plus geometric transfer operators vs. solvers using algebraic inter-grid transfer plus Ritz-Galerkin correction operators.[25,31,32] If we use geometric transfer operators, P and R—we omit the indices from hereon unless not obvious from the context—are d-linear. If we use rediscretization, A_ℓ stems from a Ritz-Galerkin finite element discretization with the same type of shape and test functions on each and every level. If we use algebraic inter-grid transfer operators, P has to be constructed such that any projection of a correction on level $\ell - 1$ is locally mapped onto A_ℓ's nullspace on level ℓ. We use BoxMG[15,31,33] to approximate such operators. The restriction is either the transpose of P or a symmetrized version of it.[15] A Ritz-Galerkin coarse grid operator $A_{\ell-1} = RA_\ell P$ is computed from the fine grid operator plus the (possibly algebraic) restriction and prolongation. Thus, it mimics the fine grid operator's impact on a coarser solution. When using rediscretization plus geometric inter-grid transfer operators one "hopes" that the resulting coarse grid operator exhibits similar properties.

BoxMG plus Ritz-Galerkin yield the same operators as geometric rediscretization on our meshes if $\epsilon \in \Omega$ is constant over the domain while it requires the absence of further PDE terms such as convection, nonlinear terms, and anisotropic material. The finer the grid the more dominant the second order term in most PDEs. Furthermore, the finer the grid the "smoother" or rarer the ϵ transitions—unless we have totally random or noisy ϵ distributions. As a result, there are often (fine grid) sections within our cascade of meshes where geometric and algebraic operators are indeed the same or very close even though complex equation terms and material are present.

Multiplicative vs. additive schemes. Multigrid solvers can be classified into additive solvers and multiplicative ones.[8] Additive solvers determine the equation system's residual on the finest mesh ℓ_{max}, restrict this single residual to all levels ℓ, and determine a correction on all levels concurrently. Finally, these corrections are then added up, subject to suitably defined prolongations. The simplest multiplicative solvers determine the residual on a level $\ell = \ell_{\text{max}}$, smooth it, compute an updated residual, restrict this residual to the right-hand side of the next coarser level, correct the solution there by applying the scheme recursively, and finally add all corrections subject to a prolongation to the fine grid estimate.

Multiplicative solvers in general converge in fewer iterations than their additive cousins, as an update on one level propagates through to other levels prior to updates on those levels. A conceptional disadvantage of multiplicative multigrid is that solver steps on the coarser mesh resolution levels tend to struggle to exploit all hardware concurrency. Cores start to idle. Any strategy yielding tasks thus can expect that these tasks exploit idling cores at one point. This is one motivation for us to focus on improving concurrency for additive schemes—if the approach works for additive solvers, it pays off for multiplicative schemes too.

4.1 | adAFAC-x

Additive solvers are often used solely as preconditioner, as they yield inferior convergence and face severe stability problems. They tend to overcorrect the solution and this overcorrection is more severe for increased number of grid levels.[2] It is thus convenient to damp the updates of the equation systems by a factor ω that depends on the grid levels. A convenient choice is to use an ω on the finest level, ω^2 on the next coarser one, and so forth. While this prevents overshooting, it tends to destroy multigrid convergence for larger systems as the elliptic nature of Equation (1) implies that any local change propagates through the whole domain. With shape functions with local support, this propagation can only be realized on coarser levels. However, exponential damping with $\omega, \omega^2, \omega^3, \ldots$ and $0 < \omega < 1$ effectively prevents changes from propagating rapidly via the coarse resolutions.

An alternative to aggressive damping is additively damped asynchronous FAC,[34] which is a solver inspired by AFACx.[35-39] The term asynchronous refers to its additivity, that is, highlights that the individual level updates can be computed without any synchronization. Additively damped denotes that a level's damping parameter is similarly computed independently of all other updates as an update for a single level is

$$u_\ell \leftarrow \underbrace{u_\ell + S_\ell\left(R\hat{r}_{\ell+1} - A_\ell u_\ell\right)}_{\text{Standard additive scheme with FAS}} - \underbrace{\widetilde{P}\left(S_{\ell-1}\widetilde{R}\left(R\hat{r}_{\ell+1} - A_\ell u_\ell\right)\right)}_{\text{Additional additive damping of correction}} . \tag{2}$$

S_ℓ here denotes the smoother approximating the impact of $A_\ell{}^{-1}$. Jacobi yields $S_\ell = \omega diag^{-1}(A_\ell)$, that is, here is where the original damping ω enters the equations. The scheme (2) projects a residual r_ℓ immediately one level further—possibly using a modified restriction $\widetilde{R}$—before it evaluates the additive multigrid term on level ℓ. When we update the solution on level ℓ, we damp our actual update with the projection of an update step of this further restricted equation. The idea is that the auxiliary equation mimics the overshooting potential of the real additive correction running concurrently through the augmented $\widetilde{R}$. To achieve this, we either apply a smoothed prolongation operator $\widetilde{R}$ to our auxiliary equation—as we use a Jacobi smoother to construct such a $\widetilde{R}$, we speak of adAFAC-Jac—or choose $\widetilde{P}, \widetilde{S}$ and $\widetilde{R}$ such that the resulting smoother resembles a BPX-like scheme.[34] The latter case uses solely P and the injection I for the auxiliary equations. We thus refer to it as adaFAC-PI.

adAFAC-x, that is, adAFAC-PI and adAFAC-Jac, are interesting additive multigrid flavors, as they use Galerkin based operator constructions twice per level. As a consequence, any inaccurate operator representation has twofold knock-on effects on coarser levels. With BoxMG and geometric operators, we have different combination opportunities to construct our operators, and it is clear that the algebraic BoxMG variant is subject to multifaceted input approximation inaccuracies if fine grid stencils are not determined correctly.

4.2 | Numerical computation of stencils in a task language

If all operators are geometric and we rely on rediscretization on every level, we can implement the multigrid scheme in a matrix-free way, once we embed the entries of the vectors $u_\ell, r_\ell, \hat{r}_\ell, \ldots$ directly into the mesh vertices. Let our program traverse the mesh. Whenever we enter a cell, we load its adjacent 2^d vertices. Each vertex holds its corresponding $u, r, \ldots$ entry, that is, one scalar per entry. Hanging vertices hold interpolated weights or zero, respectively. As rediscretization lacks dependencies on other operators, we can construct all the multigrid operator ingredients we need on-the-fly. We compute them, apply them to the local data, and immediately add the impact of the operator application (matrix-vector product) to the residual data within the mesh. Once all 2^d cells surrounding one vertex have been traversed, we can directly apply our Jacobi point-smoother, restrict, or prolongate all data.[2,5,26] We traverse our grid cells and thus accumulate the matrix-vector impact cell-wisely. For this, we require cell-wise (element-wise) stiffness matrices.

Let $\mathcal{S}$ denote a smoothing task of the multigrid algorithm, that is, a task that realizes the update of S_ℓ from Equation (2) for one particular vertex/degree of freedom only. Let $\mathcal{A}^{(\text{geo})}$ denote a (geometric) assembly task for a single element that is adjacent to the designated vertex/degree of freedom. A geometric, matrix-free implementation of multigrid then issues a series of

$$\mathcal{S}^\circ \quad \underbrace{\left(\mathcal{A}^{(\text{geo})} + \mathcal{A}^{(\text{geo})} + \ldots + \mathcal{A}^{(\text{geo})}\right)}_{2^d \text{ assembly tasks for the } 2^d \text{ cells adjacent to one vertex}} \quad + \mathcal{S}^\circ \left(\mathcal{A}^{(\text{geo})} + \ldots + \mathcal{A}^{(\text{geo})}\right) + \mathcal{S}^\circ \left(\mathcal{A}^{(\text{geo})} + \ldots + \mathcal{A}^{(\text{geo})}\right) + \ldots \tag{3}$$

tasks over the grid, that is, unknown, entries. All operators are to be parameterized over the levels. As we work element-wise, we do not evaluate each cell task $\mathcal{A}^{(\text{geo})}$ 2^d times. Instead, we set up the element matrix once, and immediately feed its impact on the surrounding u_h values in the 2^d residuals. The smoother $\mathcal{S}$ then acts on the residuals. An assembly of the inter-grid transfer operators is omitted, as we know these operators and can hard-code them. The assembly determines one stencil, that is, integrates

$$\int_{\Omega_h} \epsilon\,(\nabla u, \nabla \phi) \ \ dx = \sum_{c \in \Omega_h} \int_c \epsilon\,(\nabla u, \nabla \phi) \ \ dx. \tag{4}$$

over 2^d cells c. In our implementation, we exclusively work with an element-wise assembly where $\mathcal{A}^{(\text{geo})}$ computes the outcome of 4 over one cell c. Equation (3) has 2^d tasks that feed into one smoother though each $\mathcal{A}^{(\text{geo})}$ task which in turn feeds into 2^d $\mathcal{S}$ 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 to remove redundant tasks. We stress the entire procedure remains inherently additive however.

With explicit assembly, we run

$$(\mathcal{S}^\circ \ \ldots \ ^\circ \mathcal{S})^\circ \left(\ldots + \mathcal{A}^{(\text{geo})}_{k^2 h} + \mathcal{A}^{(\text{geo})}_{kh} + \mathcal{A}^{(\text{geo})}_{h} \right). \tag{5}$$

In Equation (5), 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 all coarse grid equations if we commit to a generating system.[30] The addition in the assembly illustrates that submatrices within this large equation system can be constructed (assembled) concurrently, as the individual cells on all levels are independent of each other.

To make a finite element discretization consistent, the assembly $\mathcal{A}^{(\text{geo})}$ has to evaluate Equation (4) over all cells consistently, that is, 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 Equation (4) for $\epsilon = 1$ and scale it upon demand. If $\epsilon \neq const$, (4) 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, that is, 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 center (Figure 2). 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 realization of the assembly, that is, it is better to make $\mathcal{A}^{(\text{geo})}$

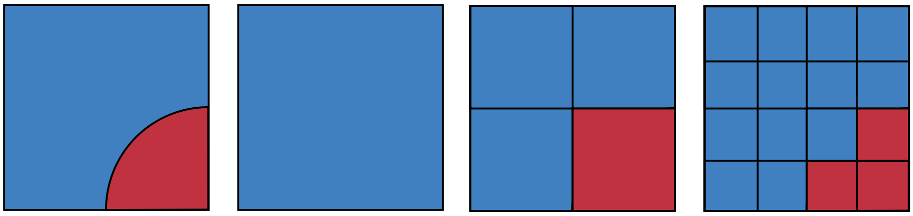


FIGURE 2 Exact material parameter within a cell (left) and a splitting of the material parameter into n^d quadrants for numerical integration (right)

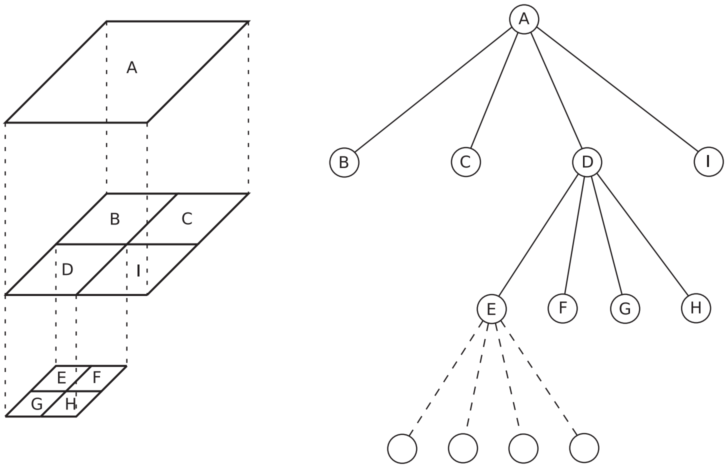


FIGURE 3 The tree nodes of a spacetree (right) span the multilevel adaptive Cartesian mesh (left) and also carry the local assembly matrices and *P* and *R* operator parts. Fine grid nodes (B,C,E,F,G,H,I) carry geometric operators, while all others carry algebraic operators. If tree nodes refine (node E), then the operators of the refined node (E) have to switch from geometric to algebraic, which in turn requires recomputations of all coarser ancestors within the tree (A and D)

act per cell and feed into the 2^d adjacent vertices rather than computing Equation (4) per $\mathcal{S}$ invocation. The latter option would effectively integrate Equation (4) 2^d times.

As we know that different ϵ distributions require different choices of *n*, it is convenient to parameterize 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 we evaluate each task once per cycle, that is, multiple times overall. Therefore, it is in the interest of the user to choose *n* as small as possible—$\mathcal{A}^{(\text{geo})}(n)$'s workload is in $\mathcal{O}\left(n^d\right)$—yet still reasonably accurate. Along the same lines, it is possible to distinguish different assembly realizations by means of their accuracy, that is, whether we run them in double or single precision.

4.3 | Stream-based matrix embedding into grid data structure

An element-wise traversal of the set of meshes $\left\{\Omega_0, \Omega_1, \dots, \Omega_{\ell_{\text{max}}}\right\}$ defines an ordering on the cells of the mesh. It yields a stream of cells. Additive multigrid's promise is that it exhibits a higher concurrency level than multiplicative schemes. Consequently, there is no order constraint on the cell enumeration within the stream; different to multiplicative multigrid where the pre-smoothing of cells within a level ℓ precede cells of level $\ell - 1$ creating causal dependencies. It nevertheless pays off to realize such a partial level ordering, as it allows us to also integrate the bottom-up residual restriction within the stream. If we run the grid traversal on a parallel machine, the cell stream is split up into multiple streams. Obviously, the cell ordering yields the exact data access pattern for the vertices,[26,40] too, but this fact is not of primary interest here.

As soon as permanent recomputation of the stencils becomes too expensive, we have to memorize, that is, store the assembly matrices. If we continue to avoid the maintenance of an explicit matrix data structure, it is natural to store the local element matrices directly within the tree cells. We 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. This approach works as long as we can guarantee that the cells are always visited in the same order (per core/rank) by the traversal. We work with our custom mesh and linear algebra implementation here. Such a constraint however would allow the scheme to be realized within other software offering matrix-free work through callbacks, too.

Once we have introduced this machinery, the in-stream storage can hold the algebraic operators $\mathcal{A}^{(alg)}$ rather than their geometric counterparts: on the finest mesh level, we make $\mathcal{A}^{(alg)}_{\ell_{max}} = \mathcal{A}^{(\text{geo})}_{\ell_{max}}$. On all other levels, we make $\mathcal{A}^{(alg)}_{\ell_{max}}$ hold the Ritz-Galerkin operator instead of a rediscretization. Finally, we can use the same implementation technique to hold algebraic inter-grid transfer operators, too. If all redundancies are eliminated—if we split up nodal operators over a vertex's adjacent cells, the distribution of the stencil is never unique—the memory footprint per cell within the stream thus grows by a factor of $\frac{3^d+2(2k-1)^d}{3^d}$. We can now hold *P* and *R* explicitly within the stream however.

The present discussion introduces a storage scheme yet ignores the causal dependencies in the computation of the stored data (Figure 3). That is, we assume that all data held in-situ is readily available; an assumption we explicitly pointed out to be wrong. A discussion on the computation rules plus the validity of all operators is handled within the subsequent section. Once we work with dynamically adaptive meshes, a task-based formalism inserts grid update tasks $\mathcal{U}$ into the task graph. These tasks either remove cells from the mesh or add cells. In the former case, they also remove entries from the stream. In the latter case, they insert entries. In a multigrid environment with algebraic operators, $\mathcal{U}$ tasks change entries of existing stencils: removing cells switches Ritz-Galerkin correction operators into discretization stencils; adding cells switches the held operators in the opposite way. Both switches imply coarser operators change that cascade, as the Ritz-Galerkin condition ensures that coarser operators depend on finer ones. This information update pattern is the subject of the discussion that follows.

5 | STENCIL ASSEMBLY VARIANTS

To gain flexibility when and with what accuracy we assemble matrix entries, we introduce per mesh cell *c* a marker

$$p\left(c\right)=\left(p_1,p_2,p_3\right)\left(c\right)\in\left\{\mathbb{N}^+\cup\{\bot,\top\}\right\}\times\{\bot,\top\}\times\{0,2,3,4,\ \dots\ ,8\}\,.$$

We reiterate that we operate in a generating system/spacetree context, that is, there are cells on each and every mesh level and different levels logically overlap. The tuples are embedded into the cell stream as headers, that is, they precede the matrix entries, and the tuple entries have the following semantics:

p_1 If the first entry holds $\bot$, we have not yet performed any integration of Equation (4) with any discretization *n*. As a consequence, the matrix entry stream (matrix linearization) does not hold entries for the corresponding stiffness matrix. If the first entry holds $\top$, our algorithm has concluded that this stencil is computed with sufficient accuracy already. The corresponding entries are found as the next 3^d or $3^d+2(k{-}1)^d$ entries within the stream. If the first entry holds a natural number $p_1(c)=n$, the matrix stream hosts the entries of the matrix, too. In this case, these entries result from an integration of Equation (4) with an n^d subgrid, and we cannot be sure yet that this *n* is sufficiently large.

p_2 The second entry holds an atomic marker. If it is set to $\top$, there is a task spawned into the task system which is currently computing a new approximation to the stencil, that is, the code is in the process of evaluating Equation (4). If the entry equals $\bot$, however, no update of the stencil is currently scheduled. The constraint $p_1(c)=\top\Rightarrow p_2(c)=\bot$ holds.

p_3 For $p_1(c)=\bot$, the third entry has no semantics. Otherwise, it encodes in which precision the matrix entries are encoded. The marker determines how many bytes we have to continue to read within the stream to obtain our element-matrices and how these bytes have to be converted into floating-point numbers.

5.1 | Delayed stencil integration

Let all cells initially carry $p_1(c)=\bot$. The classic assembly phase (5) determines all multigrid operators prior to the (first) assembly (Figure 4), and implicitly sets $\forall c:p(c)=(\top,\bot,8)$. 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 and then embedded into the stream for future use. Given a fixed, global $n>0$, a lazy multigrid code tests in every cell whether $p_1(c)=\bot$ before it runs the local assembly or matrix-vector product. If $p_1(c)=\bot$, we either integrate (4) with an n^d subgrid or compute the multigrid stencils plus the inter-grid transfer operators due to the BoxMG/Ritz-Galerkin formulation. Immediately after that $p(c)\leftarrow(\top,\bot,8)$.

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. The coarse grid assembly $\mathcal{A}_{\ell-1}$ consisting of both the construction of the element-wise operators plus inter-grid transfer operator entries of $R_\ell^{\ell-1}$ and $P_{\ell-1}^{\ell}$ is thus by definition already a series of ready tasks. They have no incoming, unresolved dependencies. They can be executed straight-away. The observation holds for both geometric and algebraic multigrid operator variants.

For additive multigrid, this straightforward lazy stencil integration works if and only if we stick to rediscretization and geometric transfer operators. It breaks down as we switch to algebraic operators, unless we prescribe the order the levels are traversed, that is, unless we ensure that the traversal of level ℓ is complete before we move to level $\ell-1$. We can weaken this statement[2,26] 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.

Observation 1. With a lazy stencil integration, we can only utilize geometric coarse grid operators unless we accept that an access to a coarse grid stencil can trigger the lazy (on-demand) evaluation of assemblies on finer levels.

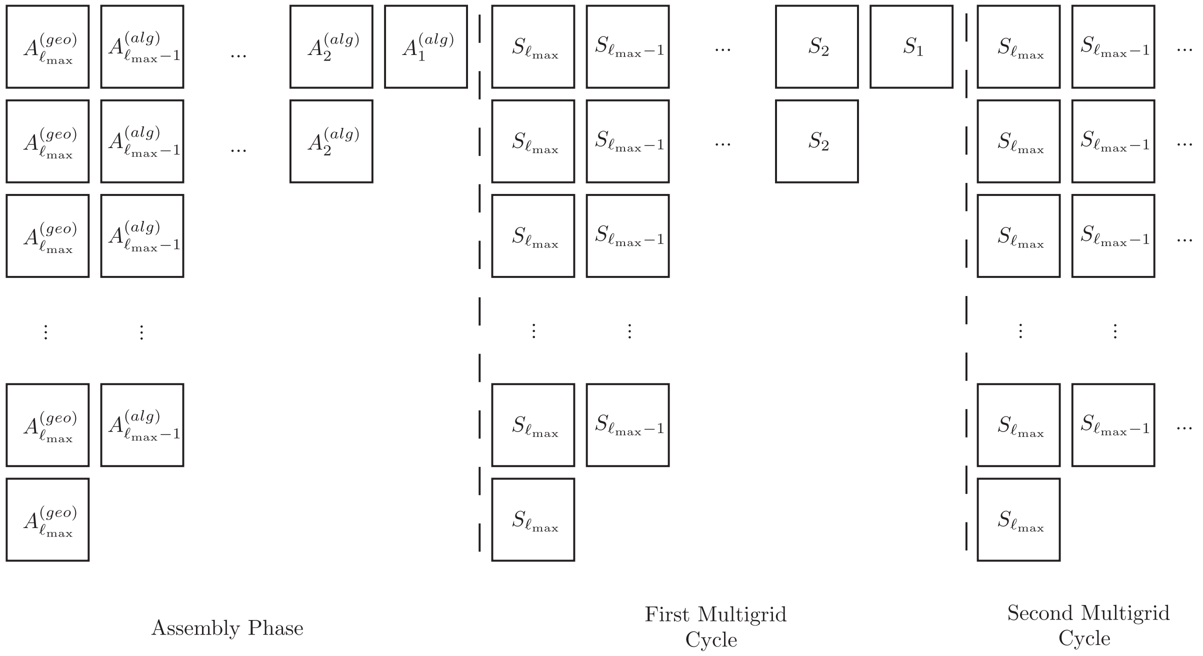

$A^{(geo)}_{\ell_{\max}}$ $A^{(alg)}_{\ell_{\max}-1}$ ... $A^{(alg)}_{2}$ $A^{(alg)}_{1}$ | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ ... S_2 S_1 | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ ...
$A^{(geo)}_{\ell_{\max}}$ $A^{(alg)}_{\ell_{\max}-1}$ ... $A^{(alg)}_{2}$ | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ ... S_2 | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ ...
$A^{(geo)}_{\ell_{\max}}$ $A^{(alg)}_{\ell_{\max}-1}$ | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ ...
$A^{(geo)}_{\ell_{\max}}$ $A^{(alg)}_{\ell_{\max}-1}$ | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ | $S_{\ell_{\max}}$ $S_{\ell_{\max}-1}$ ...
$A^{(geo)}_{\ell_{\max}}$ | $S_{\ell_{\max}}$ | $S_{\ell_{\max}}$
Assembly Phase
First Multigrid Cycle
Second Multigrid Cycle

FIGURE 4 Schematic illustration of classic multigrid's task orchestration over six cores. Time runs from left to right. $A^{(\mathrm{geo})}_{\ell_{\max}}$ denotes a stencil integration over a single cell on level $\ell_{\max}$, $A_{\ell}{}^{(\mathrm{alg})}$ denotes an algebraic operator assembly (using $A_{\ell+1}{}^{(\mathrm{alg})}$), S_{ℓ} denotes the application of one smoothing step on one cell, that is, the matrix-vector product feeding into the actual smoother. The assembly feeds into the first smoothing step, that is, has terminated before we start cycling

In this article, 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 flavor of a delayed operator assembly, where missing information input is not tolerated but resolved on-demand. Lazy integration delays the assembly and, hence, the synchronization, 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 synchronization.[1]

5.2 | Adaptive stencil integration

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

Let $\mathcal{A}^{(\mathrm{geo})}(p)$ denote the assembly of the local assembly matrix of one cell. For the evaluation of (4), it is parameterized by p, or p's first tuple entry, respectively, that is, by the numerical subsampling factor for the cell. For an adaptive stencil integration, we make $\mathcal{A}^{(\mathrm{geo})}(p)$ accept the marker plus the current local element matrix as stored within the stream if $p_1(c) \neq \perp$. For $p_1(c) = \perp$, it is solely given the marker. $\mathcal{A}^{(\mathrm{geo})}(p)$ returns a new local element matrix plus transfer operators, if required, plus an updated marker p according to the following rules:

$p_1(c) = \top$ The task returns immediately and we continue to work with the existing assembly matrix encoded in the cell stream.

$p_1(c) = \perp$ Assign $p_1(c) \leftarrow 1$ and evaluate (4) with one single sample in the center of the cell. The resulting element matrix is pushed into the stream and hence used for subsequent calculations until this task is re-evaluated.

$p_1(c) \in \mathbb{N}^+$ The task evaluates (4) over the cell of interest. It uses a $(n{+}1)^d$ sub-grid with $n = p_1(c)$ to discretize ϵ. The result is stored in a new element matrix $A^{(\mathrm{new})}$. Next, the task reads the previous matrix $A^{(\mathrm{old})}$ from the stream and compares the two matrices in the maximum element-wise matrix norm. This determines the new marker

$$p_1(c) \leftarrow \begin{cases} \top & \text{if} \quad \frac{\|A^{(\mathrm{new})} - A^{(\mathrm{old})}\|_{\max}}{\|A^{(\mathrm{old})}\|_{\max}} < C \\ n+1 & \text{if} \quad \frac{\|A^{(\mathrm{new})} - A^{(\mathrm{old})}\|_{\max}}{\|A^{(\mathrm{old})}\|_{\max}} \geq C \end{cases} \tag{6}$$

[1] Our introductory paper sketching the delayed assembly strategy for the first time labelled the whole strategy as "lazy". The term lazy however has different semantics in programming languages. We thus dropped it in favour of "delayed". Lazy, in line with programming languages, then identifies a subclass of the delayed strategy.

Finally, we keep $A^{(\text{new})}$ instead of $A^{(\text{old})}$ within the stream, that is, future work will utilize the former.

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

Observation 2. After at most $n_{\max} \cdot (\ell_{\max} - 1)$ steps ($n_{\max} = \max_c\{p_1(c)\}$), all local equation systems are valid, if 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 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.

5.3 | 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 (Figure 5). We augment our previously outlined adaptive integration such that $p_2(c) = \perp$ holds initially for all cells. Whenever we require a new stencil, we set $p_2(c) = \top$, start the computation and set $p_2(c) = \perp$ upon completion. In a synchronized setup, any check of $p_1(c)$ waits (blockingly) upon $p_2(c) = \perp$. $p_2(c) = \perp$ holds initially for all cells. As long as $p_2(c) = \perp$, either no assembly task has ever been launched or the task yielding the new, improved representation has already terminated. As long as $p_2(c) = \top$, we wait (and do meaningful other work in a reasonable task environment).

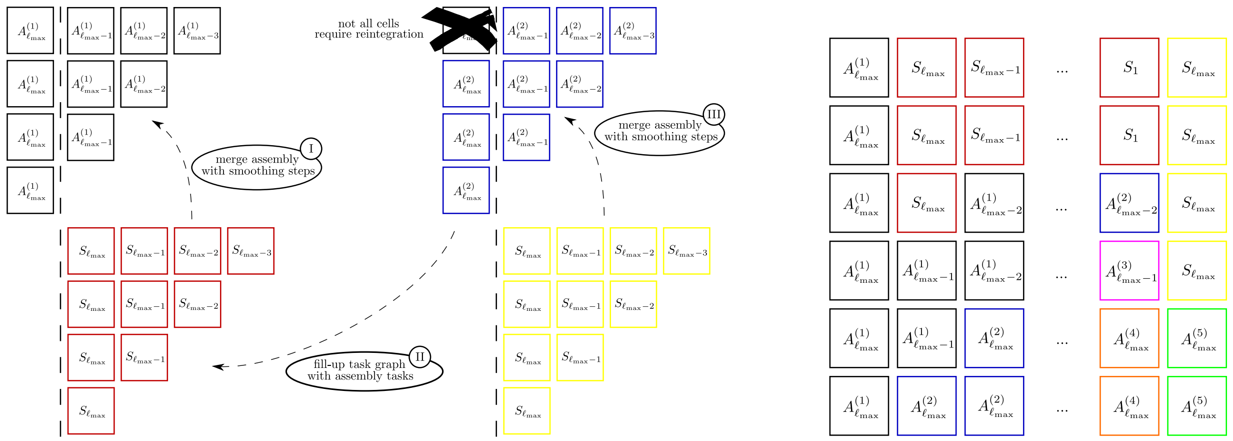
Alternatively, we can operate in an anarchic fashion and not wait upon the $p_2(c)$ entry. Other than the initial stencil computation, we launch stencil tasks using `nowait` semantics, that is, we do not wait for this subtask to terminate. Each task determines a new matrix $A^{(\text{new})}$, compares $A^{(\text{new})}$ to the previously computed one, stores the new matrix, updates the p marker and finally sets $p_2(c) \leftarrow \perp$. It realizes the third step of our adaptive integration. If the advanced integration of (4) has not terminated on time, our solver continues to use "old" stencil approximations.

Observation 3. 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 rediscretization. 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 (6) carefully: As the coarse equations are only correction equation 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.

5.4 | Vertical rippling

If we work in a Ritz-Galerkin plus BoxMG environment, 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.



$A^{(1)}_{\ell_{\max}}$
$A^{(1)}_{\ell_{\max}-1}$
$A^{(1)}_{\ell_{\max}-2}$
$A^{(1)}_{\ell_{\max}-3}$
not all cells require reintegration
$A^{(2)}_{\ell_{\max}}$
$A^{(2)}_{\ell_{\max}-1}$
$A^{(2)}_{\ell_{\max}-2}$
$A^{(2)}_{\ell_{\max}-3}$
merge assembly with smoothing steps
I
merge assembly with smoothing steps
III
$S_{\ell_{\max}}$
$S_{\ell_{\max}-1}$
$S_{\ell_{\max}-2}$
$S_{\ell_{\max}-3}$
fill-up task graph with assembly tasks
II
S_1
...
$A^{(1)}_{\ell_{\max}-2}$
$A^{(2)}_{\ell_{\max}-2}$
$A^{(3)}_{\ell_{\max}-1}$
$A^{(4)}_{\ell_{\max}}$
$A^{(5)}_{\ell_{\max}}$

FIGURE 5 Left: Construction of our delayed assembly. 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 first 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. Right: Example task orchestration/execution scheme on a six core system

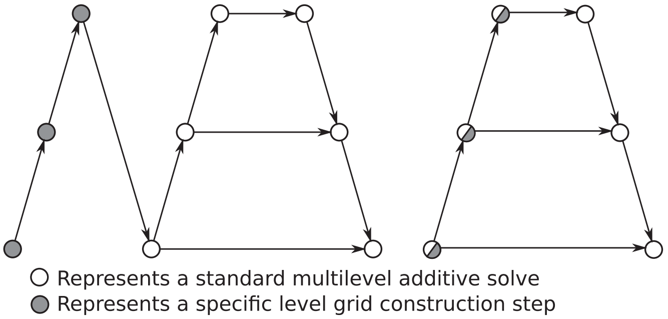

○ Represents a standard multilevel additive solve
● Represents a specific level grid construction step

FIGURE 6 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 the operators however are determined in the background, we can spawn only those tasks that might actually yield changed operators. An analyzed tree grammar[41] formalizes 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 Figure 6—formalizes how information propagates through both space and mesh resolutions if we recompute all three operators in each and every multigrid cycle.

If we employ dynamically AMR, 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 localized effect, that is, the fine grid equation system might not change everywhere—we have to change a set of affected coarse grid operators. In both cases, it is important that we reset/remove $p_1(c) = \top$ from all affected coarse grid cells. Although $p_1(c) \leftarrow 1$ obviously does the job, it might be convenient to memorize the $p_1(c) \neq \top$ last used for a cell and to reset it to this value instead. In our implementation, we permanently recompute all coarse grid operators.

Observation 4. In an additive setting, delayed operator updates ripple through the equation system, that is, the updates propagate upwards by at most one grid level per cycle. From a certain point of view, coarse grid operators on a level ℓ lag behind the fine grid operators on level $\ell_{\max}$ by $\ell_{\max} - \ell$ iterations.

5.5 | Precision toggling and information density

As the local matrix accuracy increases, the number of significant bits grows commensurately. The number of bits which carry actual information and not solely bit noise increases, too. If a local stiffness matrix was computed using small n values and there is rapid variation in ϵ over the cell, then it makes limited sense to store the whole matrix with eight bytes, that is, double precision. Many bits represent accuracy that is not really there. The same argument implies that a computation of Equation (4) can be initially done with reduced precision. If a coarse correction operator is influenced by some cells where the integration has not yet converged, it also makes limited sense to store the correction operator with full double precision. P, R, and A_{kh} could even be determined with half precision. However, deducing a priori which accuracy is sufficient is a delicate task. We propose a different approach.

Our code neglects the insight that we could use reduced-precision computing—this might be unreasonable with new hardware generations support of half precision (`binary16` or bfloats). Instead we solely use double precision compute routines, but focus on the potential of the significant bits w.r.t. the memory footprint: In all of our previous descriptions, the third tuple entry of each cell is set to $p_3(c) = 8$. This means that we use eight bytes, that is, double precision, to store each datum.

Let the operators A, P, R be stored as sequences of flexible precision values. If $2 \leq p_3(c) < 8$, each operator entry is encoded as follows: The first bit is a sign bit. The subsequent $8(p_3(c) - 1) - 1 = m$ bits hold the mantissa. The trailing eight bits hold the exponent. Our exponent is a plain `signed char` which is chosen as close as possible to the unbiased IEEE exponent. The mantissa's bits are thus nonnormalized. If a double precision value is converted into this bespoke format, then we hold it as

$$(-1)^s \cdot m \cdot 2^e \qquad 1 \leq m < 2 \ \text{ stored in 52 bits ignoring the leading 1,}$$

that is, we encode the datum as $(-1)^s \cdot \hat{m} \cdot 2^{\hat{e}}$ within the cell stream. $\hat{e} = e$ if $-128 \leq e < 127$. Otherwise, $\hat{e}$ is either -128 or 127, which is the closest value to e. We ignore/remove the fact that e is stored biased in the IEEE standard. Our $\hat{e}$ is stored as C++ `char`, that is, without any shift. If $2 \leq p_3(c) < 8$, we merely store the mantissa $\hat{m}$ as a truncated representation as m: we store its $p_3(c) - 1$ bytes in the cell stream such that the total memory footprint per matrix entry equals $p_3(c)$.

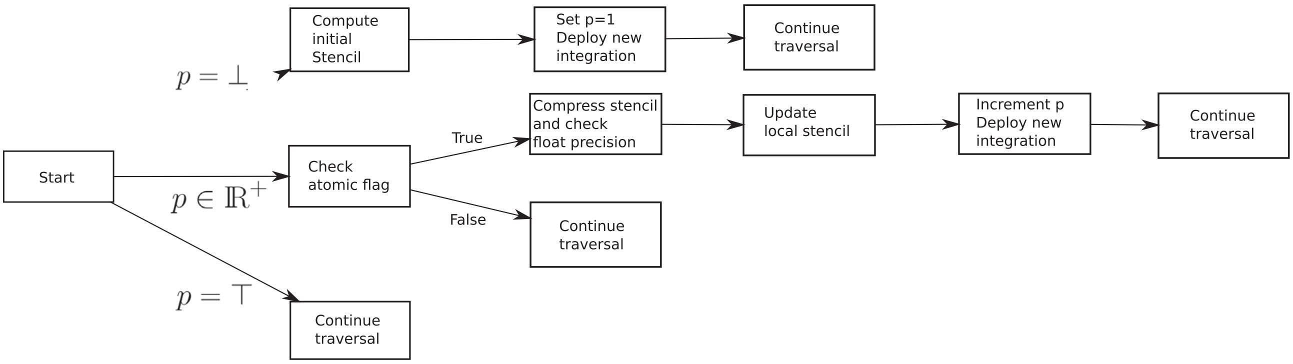


FIGURE 7 Flow chart of stencil integration logic whenever we enter a cell throughout the traversal

With this flexible storage format, we can compute the maximum error δ induced by the new format after we have computed a new stencil/matrix: Prior to any computation, we load the involved matrices from the cell stream and convert them to IEEE double precision. If an integration task computes a new assembly matrix or if the Ritz-Galerkin/BoxMG routines determine a new matrix, the matrices are converted into a representation where each matrix entry consumes only $p_3(c)$ bytes prior to their use.

Throughout the conversion we determine the maximum error δ introduced by our lossy compression. In line with the increase in the approximation accuracy n, we also increase the storage accuracy: If one byte would have made a substantial difference, then we use a higher precision. Otherwise, we stick with the same precision. For this, we usually employ two magic thresholds: one threshold guides the adoption of n, one controls the increase of the storage footprint. Both are independent of each other yet used within the same rules.

Observation 5. Our modified storage scheme increases the number of bytes spent on the matrices where it is necessary. In subdomains and/or grid levels where high accuracy is not required, that is, where the operators lack detail, we use a low-precision data format.

It is clear that this approach is of limited value in its plain version. However, it becomes powerful once we apply this idea not to the operators but to the hierarchical surplus.[5] Let $A(n = 1)$ be a local assembly matrix of a cell with one sampling point for ϵ and $P^{(geom)}$, $R^{(geom)}$ the operators resulting from d-linear interpolation and restriction. $P^{(geom)}$ and $R^{(geom)}$ can be hardcoded, while $A(n = 1)$ is cheap to compute. Instead of storing $A(n)$ or the real P and R, we store only their hierarchical surplus $\hat{A} = A(n) - A(n = 1)$, $\hat{P} = P - P^{(geom)}$, $\hat{R} = R - R^{(geom)}$. These hierarchical values typically are very small, that is, the resulting error from the compression is very small, too. Most of the cells do not require eight bytes per matrix entry to encode their (hierarchical) operators.

This hierarchical representation finally motivates us to allow for entries $p_3(c) = 0$. Whenever the hierarchical surplus of an operator equals $n = 1$ integration or d-linear transfer operators, respectively, it holds a value close to zero. If this value under the Frobenius norm is smaller than the prescribed threshold, our converted, truncated format would encode a zero. In this case, we set $p_3(c) = 0$ and skip all storage within the cell stream. We need one byte for p only. This entire process is illustrated in Figure 7.

6 | RESULTS

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 24 cores per node is available. These cores share 64 GB TruDDR4 memory. Shared memory parallelization is achieved through Intel's Threading Building Blocks (TBB), though we wrap up TBB with a custom priority layer such that we have very fine-granular control over which tasks are run when.

For the experiments, we stick to

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

on the unit square and supplement it with homogeneous Dirichlet conditions. The setup is initialized with noise from $\left[0, \frac{4}{3}\right]$.

We use two test setups. In the first one, we solve

$$\text{with } \epsilon(x) = 1 + \frac{0.3}{d} \Pi_{i=1}^{d} e^{-\theta x_i} \cos(\pi \theta x_i). \tag{7}$$

The above setup is "simple" as the analytic solution is $u(x) = 0$ for $f = 0$. Yet, the parameter $\theta \geq 0$ allows us to play around with various localized ϵ changes (Figure 8). $\theta \gg 0$ induces anisotropic behavior close to the coordinate axes and we thus know that our coarse grid operators struggle to capture the solution's behavior and thus degenerate. Furthermore, inaccurate fine grid stencils here yield wrong results and the bigger θ is the

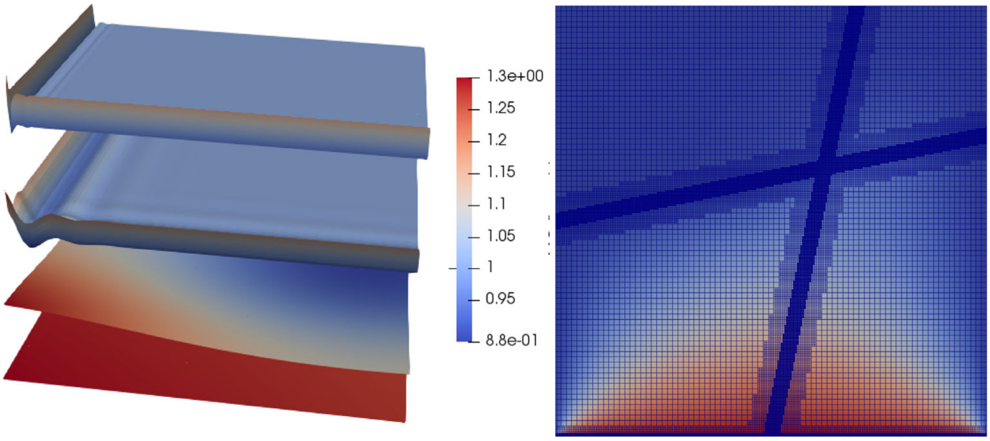


FIGURE 8 Left: Illustration of ϵ for $\theta \in\{1/8, 1, 8, 64\}$. The smallest θ value is displayed at the bottom, the biggest on the top. Right: Adaptive mesh for our two-parameter setup where the discontinuous material transition in ϵ is not mesh-aligned

greater the number of integration points n per cell. However, the bigger θ is the more localized the significant ϵ changes are, that is, nonuniform n choices are mandatory. Finally, additive multigrid tends to overshoot significantly if θ makes ϵ change with high frequency. Therefore, we can readily expose any instabilities.

In the second setup, we employ only two ϵ values, and we keep them constant over four regions of the domain (Figure 8). In two opposing regions, we choose $\epsilon = 1$. The other two regions host $\epsilon \in \{10^{-1}, 10^{-2}, ..., 10^{-5}\}$. This setup is challenging as the jump between the material parameters can be arbitrarily large and transition boundaries are not aligned with the adaptive Cartesian mesh. We cannot rely on the mesh to resolve the material changes properly. We have to rely on the local assembly matrices to accommodate the ϵ layout. As a consequence, a value of $n = 1$ within the four subdomains is sufficient. We might temporarily use $n = 2$ to decide to switch $p_1(c) \leftarrow \top$. Where ϵ changes however, we easily obtain values of $n \approx 20$ before our algorithm decides that the approximation quality of the jump is finally good enough.

All data report normalized residuals. We measure the residual and divide it by the residual in the very first cycle. The solver stops as soon as the initial residual is reduced by ten orders of magnitude. Our data usually reports the number of fine grid solution updates also called degree of freedom (DoF) updates. Overhead workload due to coarse grids is not taken into account on any axis.

6.1 | Consistency with dynamic termination criteria and starvation effects

We kick our experiments off with some studies on dynamic termination criteria. Most codes terminate the solve as soon as the normalized residual runs under a given threshold or stagnates. If we use delayed, asynchronous stencil integration, that is, 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, that is, before the right 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. 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.

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, for example, $\epsilon = 1$ or $\epsilon = 10^{-5}$, the solver's behavior changes dramatically however (Figure 9). Using a delayed assembly (stencil computation) deployed to background tasks, the solver iteration count required to reduce the normalized residual to a factor of 10^{-10} doubles compared to a solve where all operators are accurately computed prior to the solve kick-off. 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 solve 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. They thus discover that the assembly approximation is sufficiently accurate almost immediately, that is, after increasing 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

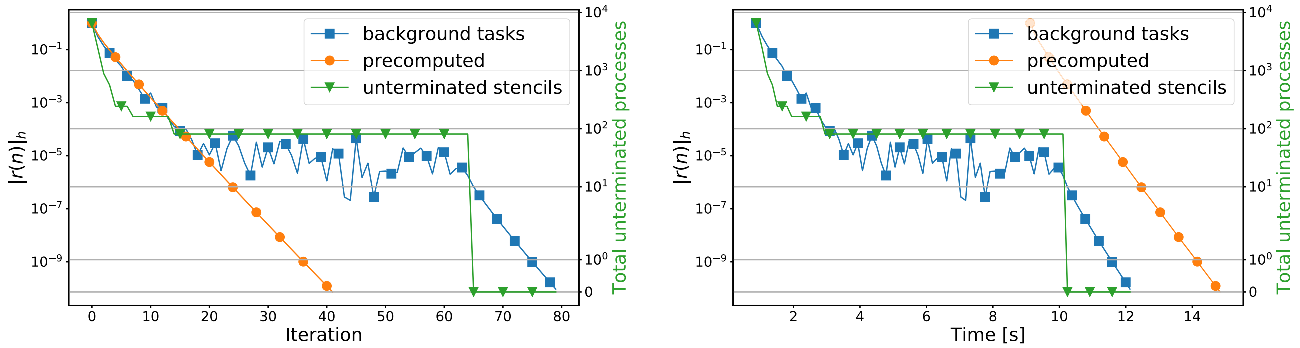


FIGURE 9 Convergence of delayed operator evaluation vs. precomputed stencils/operators per iteration (left) and against real time (right)

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, that is, do not reschedule themselves anymore, we regain multigrid convergence as we finally solve the correct system that no longer changes.

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

While our convergence considerations seem to not favor the delayed, asynchronous assembly, making the comparison with regards to real time changes the picture (Figure 9). An increased iteration count in the solver is negated due to the headstart the delayed evaluation gives the solver. The setup also highlights that 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: 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.

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

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 used coarse grid operators change both due to the delayed integration plus due to 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. On the downside, 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, too, dynamic AMR also makes the semantics of assembly matrices change: After each refinement, former fine grid discretizations suddenly become Ritz-Galerkin correction operators. With our tests, we investigate whether they still continue to push the solution in a direction that effectively minimizes the error. Our setup initially starts as a regular Cartesian mesh hosting 68 degrees of freedom with 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.

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 $\epsilon \in \{1, 10^{-3}\}$ (Figure 10). Our dynamic adaptivity criterion evaluates the solution's gradient over Ω 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 the initial residual, we terminate the solver—even though the well-defined ellipticity implies that the solver eventually will "converge back". Our benchmark compares three different solver flavors with each other. In the first variant, we immediately determine an accurate fine grid operator whenever we refine the grid. Furthermore, we stop after each refinement and reassemble all coarse operators accurately before we continue. Algebraic BoxMG operators and Ritz-Galerkin are used. In the second variant, we continue immediately whenever we refine, but we use geometric inter-grid transfer operators. The fine grid operators

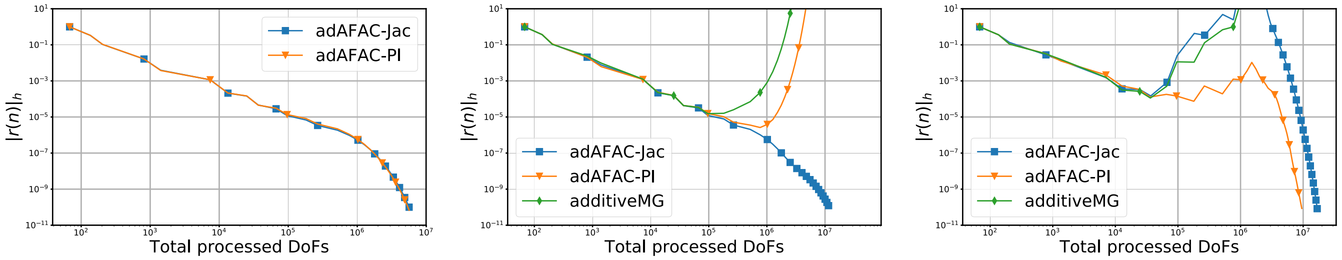


FIGURE 10 Residual plots for the jumping coefficient problem and $\epsilon \in \{10^{-3}, 1\}$. All setups employ dynamically adaptive mesh refinement either reassembling all operators accurately (left) or using delayed operator assembly with geometric operators (middle) or algebraic operators (right)

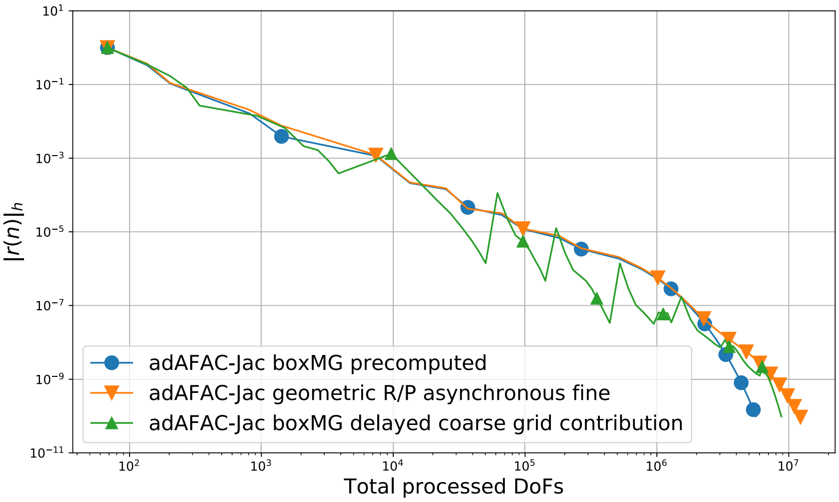


FIGURE 11 adAFAC-Jac with $\epsilon \in \{1, 10^{-5}\}$. We compare exact assembly to a Ritz-Galerkin coarse grid computation with geometric inter-grid transfer operators and a full algebraic formulation using BoxMG. The latter two options temporarily switch off coarse grids until multigrid information has rippled through

are improved through delayed integration and eventually yield improved Ritz-Galerkin coarse grid operators. In the third variant, we finally let the inter-grid transfer operators ripple, too.

The code with a complete re-assembly after each refinement step converges with a rather shallow gradient 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. If we combine geometric inter-grid transfer operators with dynamic refinement, Ritz-Galerkin operator computations and delayed integration, the adAFAC-PI solver variant suffers from a massive residual increase, while the adaFAC-Jac variant outperforms our reassembly-based solvers by more than a factor of five. If all operators are algebraic, this adAFAC-Jac suffers from significant, temporary residual explosions which eventually are recovered. adAFAC-Jac is more robust yet still not as fast as its cousin with geometric inter-grid transfer operators.

For both solver variants, our AMR 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 inter-grid 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 inter-grid transfer operators yield fine grid corrections anticipating the real material parameter behavior, but the new fine grid discretization is not yet ready to mirror them.

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

Rippling yields temporarily incompatible equation system configurations. A straightforward fix to this behavior would be an inheritance mechanisms for the number of cell integration points n: If a cell results from a coarse grid cell with n integration points, it could immediately start a first fine grid integration with $\frac{n}{k^d}$ approximation points. However, such an approach would become a caricature of the delayed approach as we would induce expensive assembly phases spread all over the solve.

We continue with an even harder setup choosing $\epsilon \in \{1, 10^{-5}\}$ (Figure 11). However, we use the insight about the maximum rippling speed as follows: We negate correction steps on a coarse level after each refinement until we observe that the next finer level operator we depend on has been updated at least once—either due to the Ritz-Galerkin recomputation or due to the delayed integration. This idea recursively applies to the next finer level if it is refined further.

We recover the stability of multigrid with an explicit reassembly though 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 F-cycle requires 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.

6.3 | Memory footprint implications

For our memory footprint studies, we return to the first test setup and study both a grid with $h \approx 0.004$ and one with $h \approx 0.038$. They are small yet allow us to showcase the impact and behavior of the proposed techniques. For different choices of ω, we track the average number of bytes per element matrix entry, the maximum number of integration points per cell that we need, and the amount of memory saved due to delayed assembly in combination with our hierarchical storage and data compression. Our data focus on the fine grid only, that is, we neglect coarse grid effects. They would blur the message and contribute to the memory footprint only marginally. The setup is configured such that the absolute error that we introduce by storing a truncated version of the hierarchical surplus is at most 10^{-8}. This is close to machine precision. As we work with the hierarchical surplus, it is reasonable to use an absolute value rather than a relative value. Finally, we use delayed operator approximation and stop the iterative computation as soon as the relative difference between two subsequent evaluations with n and $n+1$ do not differ by more than one percent anymore.

For a rather smooth parameter choice, we see that the delayed operator integration stops after it has tested the local assembly matrices for $n = 2$ against $n = 1$ variants. It cannot identify a difference exceeding one percent (Table 1). The lowest accuracy approximation is consequently used all the way through. We store the used assembly matrix relative to the $n = 1$ approximation. As they are the same here, we do not actually have to store the matrix, but it is sufficient to bookmark a one-byte marker that flags that there is no difference. A full element stiffness matrix requires $2^d \cdot 2^d \cdot 8$ double entries, that is, 128 bytes. The marker can be held in one byte. We compress the matrix data by a factor of 128.

If we use the rougher ϵ distribution in Equation (7), the delayed stencil integration increases the approximation accuracy n of at least one cell per iteration. As this monotonously grows while the average of the integration point choices remains close to 1 suggest that this is an extremely localized effect. Most of the cells are sufficiently accurate with $n = 1$, though the difference between the $n = 1$ rediscretization and the actual element matrices is not negligible anymore. We have to store the hierarchical difference and thus reduce the overall compression factor.

TABLE 1 We track the maximum number of integration points per axis n, the average number of points used over all cells, and the compression factor on a fine grid with $h \approx 0.038$ in Equation (7)

	$\theta = 1$			$\theta = 16$			$\theta = 64$		
Cycle	max{n}	average *n*	compression	max{n}	average *n*	compression	max{n}	average *n*	compression
1	1	1.00	128.00	1	1.00	128.00	1	1.00	128.00
2	1	1.00	128.00	2	1.00	118.00	2	1.07	22.64
3	1	1.00	128.00	3	1.00	118.00	3	1.15	22.64
4	1	1.00	128.00	4	1.00	118.00	4	1.22	22.64
5	1	1.00	128.00	5	1.01	118.00	5	1.29	22.64
6	1	1.00	128.00	6	1.01	118.00	6	1.36	22.64
7	1	1.00	128.00	7	1.01	118.00	7	1.44	22.64
8	1	1.00	128.00	8	1.01	118.00	8	1.51	22.64
9	1	1.00	128.00	9	1.01	118.00	9	1.58	22.64
10	1	1.00	128.00	10	1.01	118.00	10	1.65	22.64

For even higher θ choices, this reduction in compression efficiently becomes more dominant. The average n also starts to grow, that is, more cells require a local assembly matrix which differs from a simple $n = 1$ rediscretization. We still reduce the memory footprint by more than one order of magnitude however.

Observation 10. Our delayed integration in combination with compressed hierarchical storage makes the memory footprint of the solver increase over time.

We "overcompress" all operators initially and gradually approach the most aggressive compression we can use without a loss of significant bits. This is an advantageous property for algorithms with dynamic adaptivity which build up the mesh iteratively. As they start from small meshes, there is a low workload for the first few iterations. The required memory, that is, data amount to be transferred back and forth between cores and main memory, increases as the solver continues and becomes more expensive. Conversely, a finer mesh width eradicates the n-distribution observations, that is, the simulation can use $n = 1$ all over the domain. For $h \approx 0.004$, we have not been able to observe any increase in n for the three setups from above. We return to a compression ratio of 128 and an average n close to 1.

Observation 11. If the total memory footprint increases due to dynamic AMR, the delayed integration and compression in return reduce the average memory footprint per mesh cell.

6.4 | Scalability impact

We wrap up our experiments with simple single node studies—the compression and tasking paradigm has sole single node effect. The experiments run through a series of setups per tested grid. First, we assess the pure scalability of the code without any delayed integration and furthermore fix $n = 1$. Next, we prescribe $n > 1$ and make the code yield $f \in \{0.1, 0.01\}$ integration tasks per cell, that is, 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 on scalability of our idea. Finally, we run each experiment with a delayed integration twice: In the baseline, the synchronization is a preamble to the cell evaluation. In the alternative test, there is no synchronization, that is, we spawn the integration and do not wait for the result actively at any point. We work totally asynchronously. The setup is chosen such that the task spawn pattern mitigates the situation for high θ values, that is, all spawned tasks correspond to cells close to the coordinate system axes.

The partitioning with $n = 1$ yields reasonable performance (Figure 12). This obviously is a "flawed" setup from a mathematics 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 f of the cells, we indeed observe a deteriorated scalability. This is due to the fact that the high workload cells cluster along strong θ variations. 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 (Figure 13). 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.

Observation 12. The asynchronous, delayed element integration helps to regain some scalability for unbalanced setups.

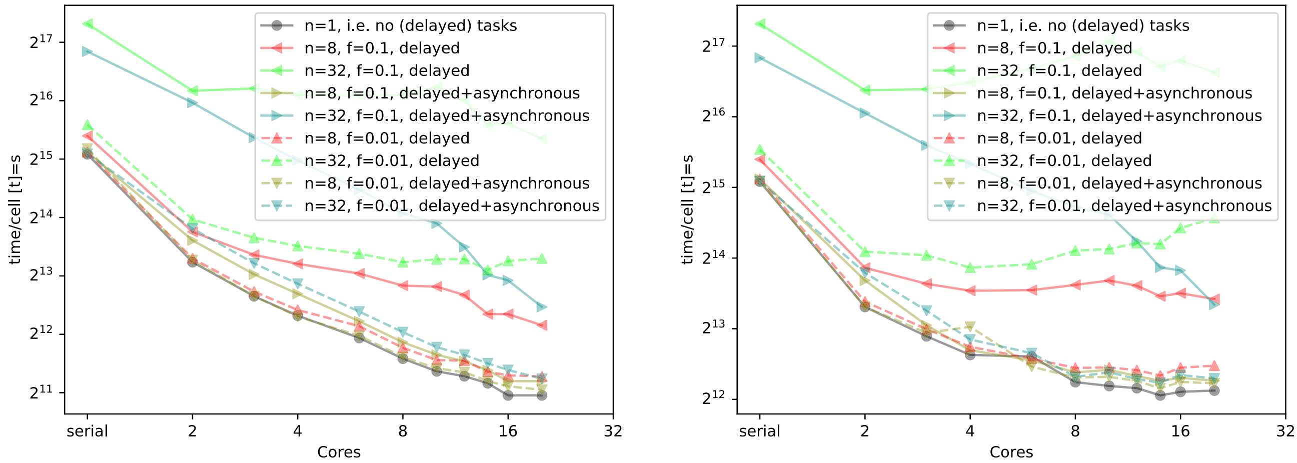


FIGURE 12 Runtime per grid sweep for one discretization with various integration/tasking configurations. Small grid with $h \leq 0.1$ (left) vs. slightly bigger grid with $h \leq 0.005$ (right)

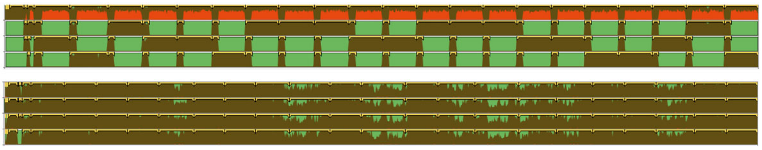


FIGURE 13 Task distribution/placement for one setup with four cores. Top: No delayed tasking is used but each cell immediately determines an improved operator before it continues. Bottom: We use an anarchic, that is, an asynchronous delayed operator integration. Brown labels denote compute work, red is spinning (active waits), green denotes idling. The graph is a zoomed-in snapshot extracted from the total execution which spans 709.5 seconds (a priori integration) vs. 428.3 seconds (asynchronous, delayed integration)

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 solver. 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 f of cells that require an improved integration as well as n. They thus study only the scalability behavior of one particular multigrid cycle. If we study the whole time-to-solution of a solver, we find that this behavior typically translates into a walltime of around 2/3 of the baseline. Baseline here is an implementation that uses the exactly same code base yet realizes the lazy evaluation pattern, that is, computes all operators prior to the first usage accurately. Walltime always comprises both assembly phase and solve phase.

7 | CONCLUSION AND OUTLOOK

Matrix assembly becomes a nonnegligible part of the overall cost of a multigrid solve within many application landscapes. It is thus important to optimize this step, too. Our proposed strategy to achieve this is three-fold: First, we abandon the idea to make the assembly fast. We instead make it more expensive, as we switch from an a priori integration of the underlying weak form to an iterative approach. In our naive implementation without any hierarchical numerical integration, this leads to redundant, repeated evaluations of sampling points. Nevertheless, we obtain an assembly that yields rough approximations quickly. It reduces algorithmic latency. The actual accurate integration is then delivered in the background of the actual computation. We hide this computational cost. Second, we introduce an anarchic variant of this delayed integration. We thus ignore previously existing synchronization points and obtain very high scalability. Finally, we propose a compressed accuracy storage format where the data footprint evolution follows the integration accuracy used. In particular around the start time, we operate with low memory footprints and, hence, low memory bandwidth demands.

Our assembly ideas face two extreme cases: Setups where expensive, algebraic operator integration is not required—well-shaped domains with constant ϵ for example give us setups where pure geometric multigrid succeeds—or setups for which an accurate operator computation is essential, as material parameters jump dramatically. For the former case, delayed, asynchronous stencil integration might introduce too much overhead, as it has to integrate each stencil at least twice to come to the conclusion that a more accurate integration is not required. In the latter case, it might yield a nonrobust implementation. Even though we use inaccurate numerical approximations initially, we obtain correct solutions with a reduced time-to-solution despite the increased computational workload for our tests. Stabililty is to be expected given that we focus on elliptic, linear problems. If properly implemented, these equations always yield the right solution agnostic of the initial guesses provided to them. Even if our anarchic, delayed approach introduces slightly incorrect initial iterates, we remain stable. Our data, however, suggest that we have to be very careful with dynamic termination criteria, and that it is very reasonable to anticipate if operator parts are completely off. Trying to correct iterates stemming from inaccurate discretizations overly aggressively can cause instabilities in solution behavior—though only temporarily. We hence propose to either fall back to geometric features within the mesh—this stabilizes the convergence behavior—or we skip multigrid updates rather than to apply updates that introduce error.

Through the lens of additive multigrid, we have looked at worst case scenarios. Inaccurate operators here should, in the theory, propagate immediately through the whole multiscale system, and there are no inherent low-concurrency phases where tasks naturally can slot in. Even a dominance of cells that do not require iterative, accurate integration has not penalized our runtime and footprint significantly. The latter results from our low precision storage. For the runtime, a single-accuracy vs. two-sample-point-accuracy integration does not make a massive runtime difference if they are all per-cell, blocked (likely cache-local) operations, and we can hence hide any compute overhead. The fact that our ideas have shown promise suggests that they will also be successful for real-world challenges and other solver types. It is a natural next step to investigate into such more complex setups—time stepping codes where multigrid is only a building block, nonlinear PDEs, or convection-dominated systems, for example—and study the impact of our proposed techniques in more detail for multiplicative solvers and more effective smoothers which might be more sensitive to inaccurate stencils. One of the most appealing strategies in an era of reducing or stagnating memory per core is our idea to store operator entries with truncated precision. So far, we use this reduced precision only to store data. On the long term, it is natural to exploit this also for mixed or reduced precision computing. This is timely as we are currently witnessing the introduction of reduced-precision compute formats due to the success of machine learning applications.

ACKNOWLEDGEMENTS

The work was funded by an EPSRC DTA PhD scholarship (award no. 1764342). It made use of the facilities of the Hamilton HPC Service of Durham University. The underlying project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE).

ORCID

Charles D. Murray https://orcid.org/0000-0002-4110-5365
Tobias Weinzierl https://orcid.org/0000-0002-6208-1841

REFERENCES

1. Dongarra J, Hittinger J, Bell J, et al. Applied Mathematics Research for Exascale Computing. Online; DOE ASCR Exascale Mathematics Working Group, 2014.
2. Reps B, Weinzierl T. Complex additive geometric multilevel solvers for Helmholtz equations on spacetrees. *ACM Trans Math Softw*. 2017;44(1):1-36.
3. Sampath RS, Adavani SS, Sundar H, Lashuk I, Biros G. Dendro: parallel algorithms for multigrid and AMR methods on 2: 1 balanced octrees. *IEEE Press*. 2008;18.
4. Sampath RS, Biros G. A parallel geometric multigrid method for finite elements on octree meshes. *SIAM J Sci Comput*. 2010;32(3):1361-1392.
5. Weinzierl M, Weinzierl T. Quasi-matrix-free hybrid multigrid on dynamically adaptive Cartesian grids. *ACM Trans Math Softw*. 2018;44(3):1-44.
6. Brandt A. Multi-level adaptive techniques (MLAT) for singular-perturbation problems. *Numerical Analysis of Singular Perturbation Problems*. New York: Academic Press; 1979.
7. Bramble JH, Pasciak JE, Xu J. Parallel multilevel preconditioners. *Math Comput*. 1990;55(191):1-22.
8. Bastian P, Wittum G, Hackbusch W. Additive and multiplicative multi-grid-a comparison. *Computing*. 1998;60(4):345-364.
9. Smith B, Bjorstad P, Gropp W. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge: Cambridge University Press; 2004.
10. Brezina M, Falgout R, MacLachlan S, Manteuffel T, McCormick S, Ruge J. Adaptive smoothed aggregation (α SA) multigrid. *SIAM Rev*. 2005;47(2):317-346.
11. Brezina M, Falgout R, MacLachlan S, Manteuffel T, McCormick S, Ruge J. Adaptive algebraic multigrid. *SIAM J Sci Comput*. 2006;27(4):1261-1286.
12. Brandt A, Brannick J, Kahl K, Livshits I. Bootstrap AMG. *SIAM J Sci Comput*. 2011;33(2):612-632.
13. Dembo RS, Eisenstat SC, Steihaug T. Inexact Newton methods. *SIAM J Numer Anal*. 1982;19(2):400-408.
14. Martínez JM, Qi L. Inexact Newton methods for solving nonsmooth equations. *J Comput Appl Math*. 1995;60(1-2):127-145.
15. Yavneh I, Weinzierl M. Nonsymmetric black box multigrid with coarsening by three. *Numer Linear Algebra Appl*. 2012;19(2):194-209.
16. Weinzierl T, Köppl T. A geometric space-time multigrid algorithm for the heat equation. *Numer Math Theory Methods Appl*. 2012;5(1):110-130.
17. Speck R, Ruprecht D, Emmett M, Bolten M, Krause R. A Space-time parallel solver for the three-dimensional heat equation. *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Vol 25. Amsterdam: IOS Press; 2014:263-272.
18. PETSc Manual Pages. https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/Mat/MATSHELL.html. Accessed February 7, 2020.
19. Gmeiner B, Köstler H, Stürmer M, Rüde U. Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. *Concurr Comput Pract Exp*. 2014;26(1):217-240.
20. Lu C, Jiao X, Missirlis NM. A hybrid geometric + algebraic multigrid method with semi-iterative smoothers. *Numer Linear Algebra Appl*. 2014;21(2):221-238.
21. Sundar H, Biros G, Burstedde C, Rudi J, Ghattas O, Stadler G. Parallel geometric-algebraic multigrid on unstructured forests of octrees. Paper presented at: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Salt Lake City, Utah: IEEE; 2012:1-11.
22. Rudi J, Malossi ACI, Isaac T, et al. An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in Earth's mantle. Paper presented at: SC'15 SIGHPC. ACM; New York, NY; 2015;5:1-5:12.
23. Eckhardt W, Glas R, Korzh D, Wallner S, Weinzierl T. On-the-fly memory compression for multibody algorithms. Paper presented at: ParCo Conferences. IOS Press; 2015:421-430.
24. Brezina M, Cleary AJ, Falgout RD, et al. Algebraic multigrid based on element interpolation (AMGe). *SIAM J Sci Comput*. 2001;22(5):1570-1592.
25. Ruge JW, Stüben K. Algebraic multigrid. *SIAM*. 1987;73-130.
26. Weinzierl T, Mehl M. Peano—A traversal and storage scheme for octree-like adaptive Cartesian multiscale grids. *SIAM J Sci Comput*. 2011;33(5):2732-2760.
27. Hackbusch W. *Iterative Solution of Large Sparse Systems of Equations. Applied Mathematical Sciences*. 2nd ed. New York: Springer; 2016.
28. Trottenberg U, Oosterlee CW, Schüller A. *Multigrid*. London: Academic Press; 2000.
29. Brandt A. *Guide to Multigrid Development*. Berlin: Springer; 1982 (pp. 220–312).
30. Griebel M. *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]*. München/Munich: Technische Universität München; 1990.
31. Dendy JE. Black box multigrid. *J Comput Phys*. 1982;48(3):366-386.
32. De Zeeuw PM. Matrix-dependent prolongations and restrictions in a blackbox multigrid solver. *J Comput Appl Math*. 1990;33(1):1-27.
33. Dendy JE, Moulton JD. Black box multigrid with coarsening by a factor of three. *Numer Linear Algebra Appl*. 2010;17(2-3):577-598.
34. Murray CD, Weinzierl T. Dynamically Adaptive FAS for an Additively Damped AFAC Variant. *arXiv preprint arXiv:1903.10367*. 2019.

35. Hart L, McCormick SF. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: basic ideas. *Parallel Comput*. 1989;12:131-144.
36. Lee B, McCormick SF, Philipp B, Quinlan DJ. Asynchronous fast adaptive composite-grid methods for elliptic problems: theoretical foundations. *SIAM J Numer Anal*. 2004;42:130-152.
37. McCormick SF, Thomas J. The fast adaptive composite grid (FAC) method for elliptic equations. *Math Comput*. 1986;46(174):439-456.
38. McCormick SF, Quinlan DJ. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: performance results. *Parallel Comput*. 1989;12(2):145-156.
39. McCormick SF. *Multilevel projection methods for partial differential equations. SIAM*. 1992.
40. Weinzierl T. The Peano software-parallel, automaton-based, dynamically adaptive grid traversals. *ACM Trans Math Softw*. 2019;45(2):1-41.
41. Knuth DE. The genesis of attribute grammars. In: Deransart P, Jourdan M., eds. WAGA: Proceedings of the International Conference on Attribute Grammars and their Applications. Springer-Verlag; 1990:1-12.

How to cite this article: Murray CD, Weinzierl T. Delayed approximate matrix assembly in multigrid with dynamic precisions. *Concurrency Computat Pract Exper*. 2020;e5941. https://doi.org/10.1002/cpe.5941