This work is dedicated to my daughters Finja Sophie and Stina Marie. I would never have been able to complete it without the constant support of my wife Marion.

Preface

More than ten years ago, the Gordon Bell Prize was awarded for a seismic calibration code [2]. According to the authors, mesh generation based upon octrees was one key feature to achieve the reported performance. Octrees look back on a long tradition of runs on the biggest machines in the world, and every year the supercomputing community continues to face new codes resting upon octree or, more general, spacetree meshes; either for solvers of partial differential equations (PDEs) or n-body codes where the original fast multipole method, being among the most important algorithms of the 20th century [24], describes spacetree meshing. *Spacetrees have been and continue to be a fundamental data structure and data organisation concept* for high performance computational science & engineering (CSE).

Much of my own research of the last six years orbits around the concept of spacetrees—with emphasis on algorithmic problems in supercomputing and less attention to supercomputing applications and high performance engineering. I thus decided to make document present eight selected papers that all rely on this particular data structure. They either tackle a particular application challenge or study, augment or efficiently realise this data structure generically. All methodological ingredients presented are integrated into one spacetree code [94]. All application-centred work relies on this code base. It is thus a valid and natural question to ask whether this document is about *'yet another spacetree code'*. It is. However, this answer comes along with the footnote that this collection of papers comprises *more than the documentation of an implementation* well-suited to write spacetree-based solvers. The algorithms and methods are of value for any spacetree or related implementation. Their integration into one code base validates that they work hand in hand. Their comparison to other approaches facilitates a classification of and differentiation to spacetree codes in general. Their application validates their usefulness and uncovers open issues in the spacetree context.

Synopsis of discussed work. The eight papers tackle different challenge flavours.

1. We discuss how we can realise the traversal of serialised spacetrees efficiently in the paper **Peano—A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids** [96]. The joint work with Miriam Mehl interprets the term efficiency in terms of low memory requirements, arbitrary dynamic adaptive grids and memory usage characteristics. While it presents ideas derived and realised in my dissertation [91], it goes significantly beyond the dissertation as it proofs the correctness of all algorithmic steps. The paper appeared in SIAM Journal on Scientific Computing (SISC) in 2011.
2. The second text appeared 2010 as part of the Parallel Processing and Applied Mathematics (PPAM) conference proceedings and is titled **A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers** [31]. This short text with co-author Wolfgang Eckhardt weakens some serialisation (inherent sequentiality) tied to the code base introduced with the previous paper, since it replaces some recursive subtree traversals with array processing.

The methodological contribution as such is incremental, but it turns out that the idea is the base for severe performance improvements in subsequent publications.

3. A second collaboration with Wolfgang Eckhardt plus Hans-Joachim Bungartz and Christoph Zenger led to the work **A Precompiler to Reduce the Memory Footprint of Multiscale PDE Solvers in C++** [17]. Published in 2010 in Future Generation Computer Systems, it augments the C++ language that allow us to compress sequences of boolean variables as well as integers and enumerations with constrained range into bit fields, to exchange subsets of C++ classes via MPI and to distinguish object attributes that are to be held persistently throughout the whole simulation from temporary ones. Whenever realisation details and small memory footprint are mentioned here, they rely on the tool DaStGen introduced with this work.
4. In **SFC-based Communication Metadata Encoding for Adaptive Mesh Refinement** [81] written together with Martin Schreiber and Hans-Joachim Bungartz we study particular properties of parallel spacetree traversals that rely on space-filling curves: Subject of interest is the linearisation of subdomain boundaries which materialises in meta data (what vertices are shared by grid fragments) that can be compressed with run-length encoding (RLE). While the compression's impact is limited—meta data are small—we can exploit the RLE property to identify data sequences that is read from a stream or sent to a neighbour as block. Vertex-by-vertex and cell-by-cell data accesses are replaced with block accesses. This speeds up the code and reduces communication time. The text became part of the Proceedings of the International Conference on Parallel Computing (ParCo) in 2013.
5. In 2014, we published the work **Block Fusion on Dynamically Adaptive Spacetree Grids for Shallow Water Waves** [93] in the Parallel Processing Letters. It is joint work with Michael Bader, Kristof Unterweger and Roland Wittmann. Subject of study are patch-based shallow water equations on dynamically adaptive grids that are able to exploit manycore architectures (Xeon Phi accelerators). In the paper, we use all previously introduced methodologies, and we apply the recursion unrolling idea to patches.
6. Joint work with Bart Verleye, Pierre Henri and Dirk Roose on a particle-in-cell application led into **Two Particle-in-Grid Realisations on Spacetrees** [97]. The novel contribution here is a Lagrangian, i.e. particle-based, formalism fused into the spacetree. All work so far tackles partial differential equations (PDEs). Furthermore, this work comprises comparisons to alternative spacetree- and SFC-based realisations and emphasises communication challenges and communication-avoiding strategies for distributed systems. Again, all fundamental algorithms come along with correctness proofs which facilitates their reimplementation. The paper is under revision for Parallel Computing.
7. As all realisations use the same code base, all apply the same programming model. This programming model is subject of discussion in **The Peano software—parallel, automaton-based, dynamically adaptive grid traversals** [92] which is submitted to SISC's SIAM CSE special issue on software. It discusses our programming workflow, the materialisation of parallelisation variants

in source code, and alternative design decisions in other spacetree codes. As such, it bridges the gap from 'What is done' to 'How is it used'.

8. The final paper results from joint work with Bram Reps. It studies **Complex additive geometric multilevel solvers for Helmholtz equations on spacetrees** [76] and is submitted to the ACM Transactions on Mathematical Software (TOMS). Again, it focuses on implementation details which become hard due to the fusion of multiple non-trivial concepts—multigrid algorithms, high problem dimensions, complex arithmetics. We are particular proud of the usage of pipelining which allows us to combine full approximation storage and additive multigrid such that each additive full approximation cycle requires only one multiscale grid sweep. The algorithm comes along with the minimal number of memory reads. As soon as we combine this property with a solver simultaneously handling multiple coupled PDEs we obtain good vectorisation results at low memory footprint; for an in-situ, dynamically generated, adaptive mesh that unfolds as in a classic F-cycle and tackles an ill-conditioned problem.

The order of the papers follows their submission date. *Their ideas are not tied to a particular spacetree code.* My work's *difference to other spacetree approaches lies in the integration* of programming model, computational as well as organisational data structures, concurrency and data exchange analysis plus algorithm correctness proofs. Other spacetree codes implement different feature sets and, in some fields, are more mature. The present assembly of ideas goes beyond that. As I link to related challenges such as proper load balancing, application realisation (stencil derivation, e.g.), IO via respective interfaces, the proposed melange of ideas does not provide a holistic solver suite. Comparisons to packages such as deal.ii [9], DUNE [10, 11], Uintah [27], AMRClaw [25], Enzo [33], Flash [30], and so forth thus are misplaced. This work is more restricted.

Outline of document organisation. This is an overview document summarising selected papers and correlating them to each other. Because of the common leitmotif and the aforementioned integration, I decided to refrain from a one-paper-after-another presentation, i.e. the text does not follow the list from the previous pages. Instead, it is structured into sections with one concept discussed per section. Within the sections, I then point out which papers make particular contributions.

The text starts with a brief introduction of spacetrees (Section 1). It is thus an excerpt and reiteration of content found in almost any of the collected papers and establishes a common language. Of particular interest is the summary of selling points of spacetrees, i.e. why spacetrees are a promising data structure for many applications, that have to be used or validated. I next pick the three papers [76, 93, 97] that tackle particular applications. Section 2 sketches what these applications are, what the overarching idea is to realise them with the spacetree, and which challenges/research questions arise from these ideas. I also highlight the major breakthroughs obtained.

As a programming model glues together application-specific code and data structures, our programming model is discussed next (Section 3). Hereby, it is important

to relate the actual coding to formal concepts such as tasks, automata and data serialisation. I also sketch how programming language extensions simplify work. The section mainly refers to core statements from [92], but implementation techniques from [17] detail how these are realised efficiently and elegantly with today's programming languages. The subsequent Section 4 discusses spacetree serialisation/linearisation. Space-filling curves here are important. A serialisation of the spacetree, i.e. its mapping onto streams, is the constitutional idea that allows for an efficient data structure traversal and elegant programming model. It however induces a total order on the data, i.e. sequential processing. This eventually leads to the question how such a linearisation fits to parallel computers. Core results and statements can be found in [91] but are formalised and proved in [96].

With an understanding of the tree-serialisation interplay, we discuss two tree decomposition approaches in Section 5. They allow us to run the serialised spacetree traversals with our programming model on distributed memory architectures. Obviously, data exchange and synchronisation from an application's point of view have to be picked up in this section as well. One key idea to distribute a spacetree among different ranks and yet continue with a serialised traversal is to apply recursion unrolling. Publication [92] acts as blueprint and source for this section which is enriched by some ideas that arise from the application-centred paper [97]. One-step recursion unrolling is extended in Section 6 into recursion unrolling of arbitrary depth. We apply it to subtrees that encode regular Cartesian subgrids. It can transform depth-first ordering into breadth-first ordering. We then can reorder the latter along the lines of the well-known red-black colouring. This yields traversals that parallelise on shared memory architectures with a bulk synchronous programming model. While the core idea of the shared memory parallelisation is introduced in [31], it is made an enabling technique for patch-based applications running on manycore systems in [93]. In Section 7, I bring space-filling curves serialising the tree, shared and distributed memory parallelisation as well as recursion unrolling together. The combination of all three ideas facilitates efficient data transfer—either from processor to memory, or from node to node, or from memory to manycore chip through several memory controllers. Core statements result from [81].

We close the overview in Section 8 where we pick up the three major applications (use cases) again, reiterate the spacetree's selling points at hands of these applications, and highlight how the algorithmic ideas introduced so far help us to answer the research questions that arose from the applications. The text closes with a classification of spacetree codes in general, i.e. puts the present concepts into perspective to other codes, and raises some research questions that had to remain open.

How to read this document. Central ideas and terms per paragraph are set in italic. They highlight the red thread through the text. The core statements from the discussed papers are set in boxes. They also come along with links to the respective publications. I do not excerpt all core statements. Only statements referring to the grid management, programming model and high performance computing (HPC) are picked up.

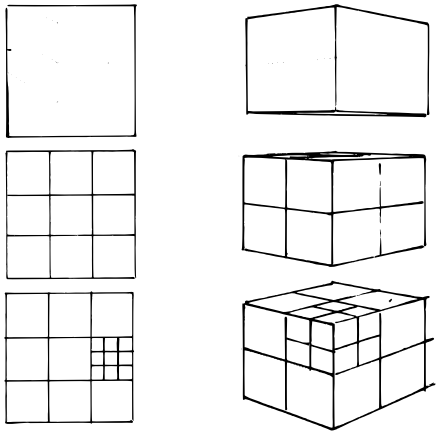
Contents

1 The spacetree meshing paradigm

All discussed papers rely on the concept of spacetrees. A spacetree is constructed as follows: We embed the computational domain Ω, subject to a partial differential equation (PDE) or a particle-based formalism, into a hypercube that is axis-aligned with the Cartesian coordinate axes. This geometric primitive is called *root* of the spacetree. We cut the primitive equidistantly into k parts along each coordinate axis and end up with k^d new d-dimensional cubes. d is the spatial dimension of $\Omega \subset \mathbb{R}^d$. Each of the k^d primitives is a *child* of the root. We finally determine for each child recursively and independently whether to continue to refine or not. The child-parent relations define a directed graph, a partial order, on the cubes. *The set of all cubes together with that order is the k-spacetree.*

> Core statement from [96]: The k-spacetree construction describes a generalisation of the well-known quadtree/octree concept.

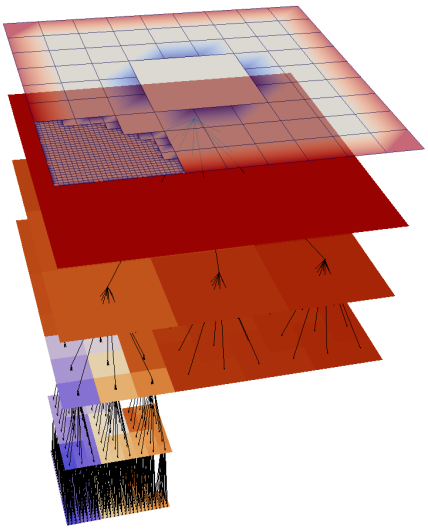
k-spacetrees as such form a class of a popular data structures in scientific computing—[2, 4, 6, 13, 22, 23, 49, 79, 84, 85] is far from a comprehensive list of examples. The term generalisation is twofold. It refers to the spatial dimension d—an eponymous property for quadtrees and octrees [78]—and it refers to the subdivision count. *The term spacetree neither fixes the dimension nor the subdivision factor.* For the present papers, it is notably inspired by [34] where it is used to highlight the d-independence, while [50, 51] and [66, 73] restrict themselves in a similar context to $k = 3$ and derive Latin identifiers. They are tongue twisters. We therefore stick to a generic nomenclature.



k-spacetrees with $k = 3$ in two dimensions (left); $k = 2$ yields an octree in three dimensions (right). From [91].

All papers in the present collection start from $k = 3$. $k = 3$ stems from the fact that we use the Peano space-filling curve (SFC) [3, 77] to design the underlying grid management. This SFC as well as its unique properties are subject of study in Sections 4 and 7. Yet, most algorithmic statements besides the grid's storage on streams in Section 4 hold for arbitrary $k \geq 2$. While we use the term k-spacetree to emphasise the generality but at the same time use $k = 3$ for all experiments, the SFC discussion in [91] already emphasises that any odd k works out for the present grid management algorithms without modifications. Furthermore, Sections 5 and 6 introduce techniques that conceptionally rewrite a $(k = 3)$-spacetree into a spacetree with different subdivision properties if certain conditions are fulfilled. In Section 8, we extend this fact to patch-based discretisations. In accordance with [87, 90, 98], we finally note that odd k are beneficial for cell-centred discretisations, as cell centres of a cell coincide with finer cell centres once an element is refined.

One advantages of the Peano space-filling curve motivating $k = 3$ is its *straightforward extension to arbitrary spatial dimensions* d—a property it shares, to the best of our knowledge, exclusively with the Morton order [3, 72, 77] (cmp. Section 4). Spacetrees can be applied to any d of $\Omega \subset \mathbb{R}^d$ anyway. While the discussed papers besides [76] stick to $d \in \{2, 3\}$, all algorithms work for any $d \geq 2$. Algorithms for $d > 3$ are of use in the context of space-time discretisations [32, 36, 37, 52, 53, 56, 57, 61, 62, 69, 70, 82, 95] or multi-parameter settings tackled by systems of partial differential equations [76]. To avoid confusion with k-d-trees, a generalisation of space partitioning to non-equidistant domain subdivision and anisotropic refinement, the spatial dimension d is not picked up in the notion explicitly.



Spacetrees yield a cascade of ragged Cartesian grids. From [93].

Spacetree-grid equivalence. Each d-dimensional cube within the spacetree has a *level* being the minimal number of refinement steps required to derive the cube from the root. All cubes of the same level have the same size. They are non-overlapping, as the children of one cube do not overlap. They are aligned with each other, as k is invariant and we cut equidistantly. They can be disconnected, as the refinement decision is made independently for each cube. All cubes of one level ℓ yield a computational grid $\Omega_{h,\ell}$. It can be a *ragged Cartesian grid* not filling in the whole root. The cubes are either *refined* or *unrefined*. We call the latter *leaves*.

> Core statement from [96]: A k-spacetree yields a cascade of ragged Cartesian grids embedded into each other. The union of all leaves of the spacetree yields an adaptive Cartesian grid Ω_h. It is flat, i.e. the cells have no overlap. The union of all spacetree nodes yields a locally refined multiscale adaptive Cartesian grid. Cells here do overlap.

A spacetree is a formal description for a special type of an adaptive Cartesian multiscale grid. The construction process makes the grid fall into the class of block-structured adaptive mesh refinement (SAMR). Since we stick to that special class of grids, we have two equivalent formalisms at hand to discuss properties and algorithms of the grid: a tree language and a grid language.

Selling points. All spacetree codes share similar advantageous properties.

- They facilitate (dynamic) **adaptivity**. Their construction maps refinement or coarsening onto adding or removing nodes from the tree. Adaptivity is one

of the state-of-the-art requirements of grid-based solvers today [29]. It allows computers to spend memory and effort where they pay off most.

- They facilitate **in-situ mesh generation**. Given a description of the computational domain and its bounding box, their construction allows to load the description into the compute nodes and to generate the mesh there on-the-fly rather than to preprocess the mesh out-of-core and then upload the mesh into each node. In-situ mesh generation is important on today's supercomputers suffering from IO restrictions [29].
- They facilitate **dimension-generic programming**. Their geometric elements fit to d-parameterised tensor product formulations. At the same time, the orthogonality of the underlying element coordinate system allows the combination of different integrators (time integrator in one direction and spatial discretisation along the other directions, e.g.) easily. Dimension-generic programming renders the realisation of multipurpose codes—mathematical prototype vs. application domain production code—more economic.
- They facilitate **geometric multiscale algorithms**. Their construction describes a hierarchy of grids and it is straightforward to exploit this hierarchy for solvers or multiple physics. Multiscale algorithms are among the most efficient linear algebra solvers. Multiphysics codes are among the great challenges today [29].
- They facilitate **domain decompositions**. Their construction describes a hierarchical decomposition of the computational domain. This built-in decomposition can be used on shared and distributed architectures to realise parallelisation. Domain/data concurrency is the dominant parallelisation strategy today and will remain important—though not the silver bullet of parallel programming—with the growth of concurrency in hardware [29].
- They facilitate **efficient algorithms** because of their structuredness.

The latter statement is a dogmatic claim that has to be substantiated.

2 Case studies

Three case studies give examples where and how spacetree-based algorithms can push applications forward. The other way round, the applications raise questions alike 'how can this be realised within a spacetree elegantly with good performance'.

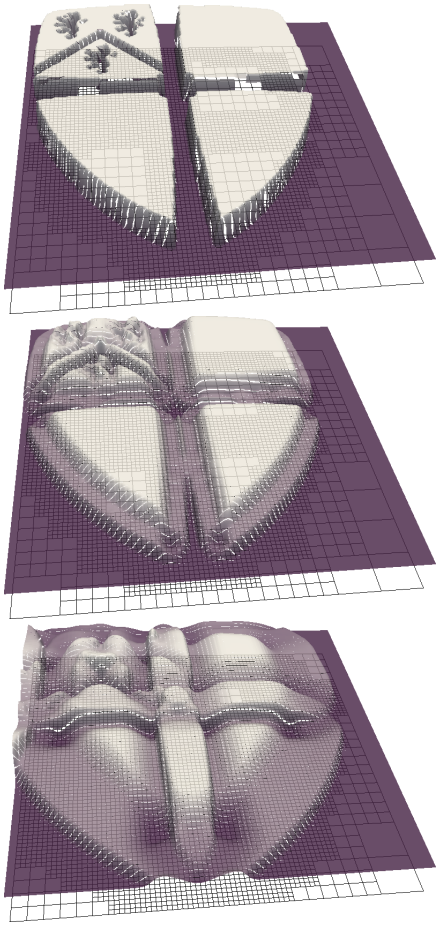
A shallow water simulation. In collaboration with the group of Michael Bader (Technische Universität München) we studied shallow water equations (SWEs)

$$\partial_t \begin{bmatrix} h \\ h\ u \\ h\ v \end{bmatrix} + \partial_x \begin{bmatrix} h\ u \\ h\ u^2 + \frac{1}{2} g\ h^2 \\ h\ u\ v \end{bmatrix} + \partial_y \begin{bmatrix} h\ v \\ h\ u\ v \\ h\ v^2 + \frac{1}{2} g\ h^2 \end{bmatrix} = S(t,x,y) \qquad (1)$$

describing a solution triple (h,u,v) over space (x,y) and time t on spacetree grids. h is the water height, u and v velocities. Using spacetrees in this application area is

common practice. Early papers ([14] for example) already describe a spacetree-like methodology. Our code implementation fragments from [87].

We start from the observation that *spacetree cells should not act directly as finite volumes for the underlying finite volume scheme*: for the required degree of resolution, very deep spacetrees would be required while the computational load per finite volume is limited. Our work proposes hence to *embed patches*, i.e. small regular Cartesian grids, into the spacetree leaves. They define the finite volumes and are augmented by a small halo/ghost layer. Thus, the spacetree describes an overlapping domain decomposition and acts as organisational data structure. On the patches, we can invoke optimised kernels (operations) to solve the Riemann problems with uniform vectorised stencils [5, 7, 68]. Benchmarks with these kernels reveal that *patches with a big number of finite volumes yield good performance*. Yet, *large patches constraint the adaptivity*: for a given memory footprint, shock surfaces can not be resolved anymore as accurately. The total memory requirements easily exhaust a node—in particular low-memory nodes such as Xeon Phis—and a high MFlop rate faces that fact that with big patches more work is invested than actually required numerically. We face a dilemma. The central questions raised by paper [93] thus read:



Shallow water code in action acting on the Durham code of arms as toy problem. From [93].

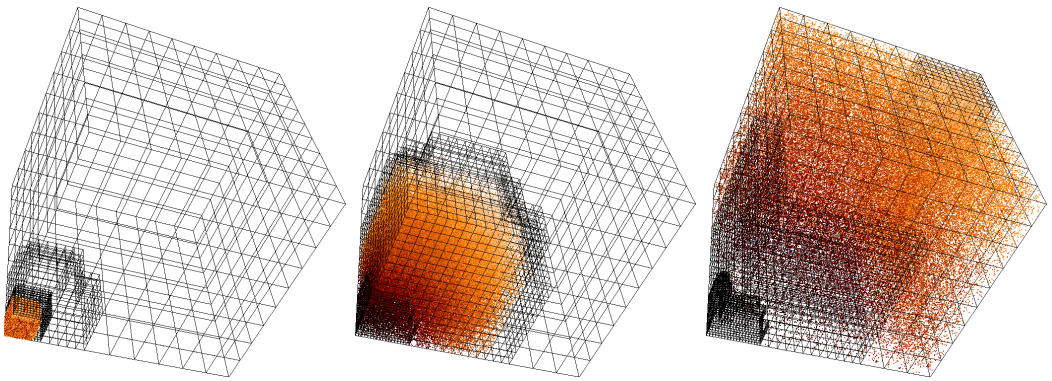
1. How can we elegantly support patches embedded into a dynamically adaptive spacetree without exposing technical details such as adaptivity management and thus requiring existing kernels for regular grids to be altered.
2. What are reasonable patch sizes for the present challenge that fit to current hardware architectures?
3. How can we compromise between aggressive adaptivity and high-throughput, regular patches used as elementary building block in the grid?

The paper relies on the multicore parallelisation strategies introduced in Section 6 and fuses them with optimised kernels written for regular grids (patches) of the group of Michael Bader [5]. Its seminal new contribution results from a combination of these patch-based solvers with the spacetree. The code starts from one patch embedded into each leaf. As soon as assemblies of leaves are identified which can be fused into one large regular grid, such a grid is embedded into the coarser levels of the spacetree and the code continues to work with this large, regular data structure

until dynamic adaptivity forces it to break up again. The *on-the-fly patch fusion* relies on an analysed tree grammar very similar to ideas from [31] used in Section 6.

> Core statement from [93]: To the best of our knowledge, this is one of the first publications that were able to exploit very large vector registers plus the Xeon Phi's many cores for unconstrained dynamically adaptive grids.

A particle-in-cell code. In a collaboration with Bart Verleye, Dirk Roose and Pierre Henri (KU Leuven and the Flanders ExaScience Lab), we studied a classic particle-in-cell code simulating space weather. Particles are embedded into the computational domain hosting a partial differential equation (PDE). They determine the right-hand side of the time dependent PDE. The PDE in turn accelerates the particles. While the particles do not interact with each other, interaction happens through the PDE solution.



Particles explode in a domain and the adaptive grid follows the explosion. From [97].

Particle-grid methods are well-established in the spacetree community. We refer to the many fast multipole algorithms [24]. These are upon the best-scaling codes in the world, but *our challenge is significantly harder*. Since particles do not interact, the arithmetic intensity of the particle update and move phase are negligible. They reduce to a 'run through them once and then sort them into the right grid cell' problem. 'Unfortunately', we study setups with superthermal particles. CFL-type mesh width constraints do not apply then anymore globally: *very few particles are allowed to move with arbitrary speed and thus may jump over several cells per time step*. They tunnel. We hence have to *sort all particles globally per time step while the step has vanishing arithmetic intensity*. The central questions raised by paper [97] thus read:

4. How do we store particles within a spacetree such that grid-particle and particle-grid mappings can be evaluated fast?
5. Can we update this storage scheme quickly if the particles move?
6. Can we make the code scale even if some particles tunnel?

The paper proposes to decompose the spacetree among all ranks and to hold the particles on the respective ranks with links from the cells or vertices, respectively,

to particle lists. It studies two realisation schemes (and compares to two further schemes) together with update mechanisms for this associativity and presents correctness proofs. The last research question picks up the fact that we require global sorting per time step and necessarily run into latency issues to synchronise all ranks: Even if no particles tunnel, each rank requires a notification of all other ranks that no particles tunnel into its domain. *The paper again exploits the spacetree's multi-scale nature and analyses maximal particle velocities on all scales to deduce where parts of the synchronisation can be skipped.*

> Core statement from [97]: To the best of our knowledge, this is the first publication introducing communication-avoiding algorithms for grid-particle codes where particle velocities are not constrained.

A multigrid Helmholtz solver. In a collaboration with Bram Reps (University of Antwerp) we studied the dynamics of p quantum particles described by Schrödinger equations. Following the seminal work of [88], these high-dimensional problems can be reduced onto a system of stationary p-dimensional PDEs

$$\begin{pmatrix} H_{11} & A_{12} & \dots & A_{1c} \\ A_{21} & H_{22} & & A_{2c} \\ \vdots & & \ddots & \vdots \\ A_{c1} & A_{c2} & \dots & H_{cc} \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_c \end{pmatrix} = \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_c \end{pmatrix}$$

with the Hs representing Helmholtz equations

$$[-\Delta - \phi(\rho_1, \dots, \rho_p)]\, \psi(\rho_1, \dots, \rho_p) = \chi(\rho_1, \dots, \rho_p).$$

We rewrite the system in a Jacobi-solver manner due to an extraction of the diagonal blocks into an iterative scheme. Our algorithm then solves these p-dimensional problems—they are called channels—one after another within an outer iteration loop. Efficient solvers for the Helmholtz problems thus are essential. Efficiency comprises both the classic computer science notion of efficiency as well as the question how to tackle their indefiniteness. The paper proposes to rely on *complex scaled grids [75] where mesh and Helmholtz term are multiplied by a complex value* to tackle the indefiniteness. While such a shift changes the solution of the underlying equation system, only surface integrals around the computational domain are of interest to our application. They are called field field maps [26, 76] and are not significantly polluted by the shift.

Nucleon and electron distribution probabilities for Deuterium that result from various Helmholtz solves. From [76].

Multigrid solvers are among the fastest solvers known. Our code focuses on additive schemes as they map directly onto tree traversals and are reasonably robust. Experiments however reveal that plain additive multigrid is insufficient to solve problems with a Helmholtz term. We thus extend the spacetree-based solvers from [50, 66, 71, 73] into a hierarchical basis [48] and a BPX-variant [12]. Because of the high dimension p, we require an F-cycle type solver unfolding an adaptive grid throughout the solve phase. This allows us to end up with convergence rates close to multiplicative schemes for some problem setups. Grid unfolding yielding different system matrices per iteration benefits from matrix-free solvers. For those, full approximation storage (FAS) schemes that hold a solution representation on each resolution level are advantageous as they allow to work with a hierarchical generating system [48] and the same stencils on each level. We show in simple proof via contradiction on the data flow graph that *a combination of the objective to have one multigrid cycle per tree traversal and FAS does not work straightforwardly.* It requires us to use pipelining [38, 40–42]. Furthermore, a plain implementation *suffers from the low arithmetic intensity per cell.* The central questions raised by paper [76] thus read:

7. How can complex-valued arithmetics be supported in the spacetree world (even the mesh widths might be subject of complex scaling)?
8. How can pipelining in the implementation help to solve the Helmholtz problems efficiently, i.e. with a read-once update-once policy per unknown?
9. How can we obtain high vectorisation efficiency for the simple stencils?

In this particular application, parallelisation does not play a major role. The channel decomposition system yields embarrassingly parallel systems to be solved. Because of an unsaturated memory bandwidth usage, we propose to introduce helper variables. This helps us to realise one unknown update per sweep including prolongation in a multigrid sense and injection in a FAS meaning. As we fuse the solve of multiple Helmholtz problems and integrate some coupling matrices (A_{ij}) into the F-cycle, we increase the arithmetic intensity per cell. Though this reduces the level of concurrency—it fuses channels—the number of channels remains suitably large to exploit medium-sized parallel computers.

> Core statement from [76]: To the best of our knowledge, this is the first implementation with correctness proofs that combines FAS, BPX or additive multigrid with F-cycles as well as complex-valued arithmetics on a spacetree. It works without explicit matrix assembly and realises pipelining such that one additive multigrid/BPX cycle is evaluated per tree traversal.

3 Programming paradigms

Traversal concept. With a sketch what is solved on which tessellation, a central computer science challenge is *how the algorithmic ingredients and data structures*

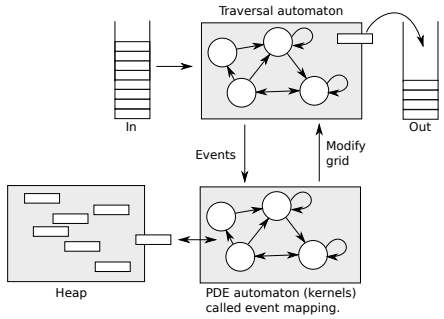
interplay. What is the programming paradigm? In [92] we distinguish *two different concepts* to realise a code within a spacetree environment. One can either provide the application code with the opportunity to access any grid entity at any time. This is a *random access model* (RAM). Or one can *prescribe the order all mesh entities are processed* and, at the same time, constrain which data is available at which time. The latter paradigm can be formalised by a traversal automaton which calls application routines in a predefined order and hands them over a well-defined set of grid entities. We call these calls *events* and formalise this interplay of traversal and application codes in terms of interacting automata in [92], while older, outdated lists of events can be found in [19, 71, 91]. Events are for example 'use/read a particular vertex for the very first time throughout a traversal' or 'descend from level ℓ into level $\ell + 1$ for a cell'. For RAM, all responsibility for a proper grid walk-through is left to the user, and the spacetree comes along as library providing information such as multiscale relations or adjacency information. Our papers rely on an inversion of control. A basic framework implements grid storage and traversal as automaton, and the application plugs into this framework to realise the application tasks. This realises a separation of concerns where grid and data management as well as grid traversal are hidden from the application-specific code.

Spacetrees are popular and mature in the fast multipole method (FMM) for which a *third, hybrid programming model* is important: the *dual tree traversal* [28, 54, 89]. 'Dual' refers to the traversal; not to be confused with a dual tree as used in [97]. The dual tree traversal makes two automata traverse the tree. Per state of the first automaton, the other runs through the whole tree once. Application code fragments plug into changes of the automata state tuple. Any combination of grid entities—automata states comprising cells plus adjacent vertices—thus becomes available at a time, i.e. no constraints apply to the data access, while the sequence of data availability is prescribed by the automata. Obviously, such a traversal that is quadratic in terms of spacetree nodes can be speeded up if the code filters out, i.e. skips, automata combinations that are not of use (cmp. to local essential or pruned trees). Within this mindset, our automata-based traversal applies a particular filter: it restricts to automata tuples where both states are equal.

> Core statement from [92]: Our codes rely on an automaton traversing the spacetree. They plug into the automaton's state transitions (events) to realise application-specific behaviour; change properties of the grid entities for example.

It is obvious that the automaton formalism realises a concept well-known under several names besides 'inversion of control'. The wide-spread concept of an iterator also aims to hide the realisation and the traversal order of the underlying container from the iterating code fragment. In the object-oriented world, the visitor design pattern [35] realises the same idea, though less formalised and with the opportunity to intervene with the traversal. Perhaps the most illustrative name for the programming paradigm is the *Hollywood principle* summarised as 'don't call us we call you'.

From a supercomputing point of view, it is important to emphasise that the automata realise a functional programming concept also realised by higher order functional programming languages or via callbacks (plug-in points) in many scientific software packages. The hiding of how the grid is traversed and stored not only hides away complexity, it also hides away where events are called in a distributed environment. The paradigm is *capable to hide concurrent event calls*, and it allows the underlying realisation to reorder events and optimise data accesses regarding the memory hierarchy, e.g.



The multiscale grid is traversed by one automaton reading an input stream and writing an output stream. Application-specific data might be embedded into the streams. The automata transitions (such as 'we enter a cell') are passed via events to a application code which can be understood as a second automaton. Events are enriched by corresponding application-specific data from the stream. The application code might interact with a heap and is allowed to trigger grid transitions such as 'refine area around the traversal automaton's current cell'. From [92].

To facilitate the latter features, it is important that the (partial) temporal order of the plug-in points is well-defined. We have to state which events are called for which grid parts in which order. The order is tight to the strategy how to run through the spacetree. The latter is subject of discussion in Section 4. A comprehensive list of events is presented in [92] or the code documentation. What seems to be mandatory to present in a programming paradigm context is a central observation from [92]: Once an partial event order is fixed and the application code has plugged into the events—often we refer to these realisations as (computational) kernels—we may postulate:

> Core statement from [92]: Event-based applications can be read as task graphs. The tasks are the realisations of the events, and the partial order between the events establishes task dependencies.

Data storage. Spacetrees are an abstract data organisation concept yielding also a computational grid. For any application, this grid has to be filled with actual data. We have to assign unknowns to the grid entities. Here, the application [93] differs fundamentally from [76] while the work in [97] is a hybrid (cmp. Section 8). One variant is to embed all application-specific data directly into the data structures describing vertices, cells and traversal automata states. As the automaton runs through the grid, it delivers corresponding grid entities and, with these entities, provides the application-specific data to the application kernels.

> Core statement from [92]: Application-specific data can be embedded into the linearised spacetree. In this case, the spacetree acts as both organisational data structure and compute data.

With the work of [87] enabling [93, 97], we emancipate from this data embedding and also allow codes to store their data in a map. The grid entities' spatial position plus their level then act as key to this map. We call this map the *heap* compared to a stream- or stacked-based approach where data is embedded into the spacetree data (cmp. Section 4 for a motivation of the term 'stack'). In practice, it is advantageous to cache the map index directly as index embedded into the grid vertices or cells, respectively, rather than to recompute it. Conceptionally, the heap concept picks up the early work of [49] combining hash maps and the Hilbert space-filling curve. It is a delicate, problem-specific question whether to run for heap-based data storage or to embed the data directly into the grid.

> Core statement from [92]: Because of heap storage, data of arbitrary, changing cardinality can be assigned to grid entities.

Disadvantages are that the heap may lead to a scattering of data accesses, that the maps induce a memory overhead, and that the handling is slightly more complicated than a plain embedding of data into the stream. In return, we face new opportunities: The particle handling from [97] would not be possible without heaps, as the number of particles per cell may change. The shallow water code in [93] embeds regular grids of different size into different grid levels. And all algorithms that do not need to run through all data in each grid traversal benefit.

Besides data persistence and programming paradigms, the work of [92] also sketches typical development workflows and some development tools that we ship with the grid core routines. All of those are written in C++ with an emphasis on documentation, automated unit tests, the augmentation of all routines with meaningful assertions and a rigorous separation of concerns. This helps to achieve high code quality from a software engineering point of view. However, it becomes obvious that C++ is not the optimal choice for our high performance computing challenge flavours. Notably problematic are the high memory overheads induced by C++'s realisation of inheritance, padding, the mapping of bits (booleans) and bitsets onto integers and the inappropriate support of the object-oriented programming paradigm through MPI.

Data modelling. Our codes thus rely on the tool DaStGen [17] to model all involved structs and classes as well as involved MPI messages which are mapped onto classes as well. DaStGen is as simple C++ precompiler—C++ class attributes are annotated with additional meta data—that

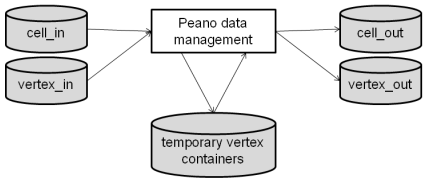
- maps bits and bitsets onto one big integer acting as bitfield. Thus, only one bit per boolean is required. Support for integers where the range is known is available as well: If an integer is annotated to hold only values between 1 and 8, only three bits are required for this integer.
- generates user-defined MPI data types such that C++ classes can directly be used with MPI while only those attributes of a class are exchanged that are explicitly marked. All other attributes are not sent through the network.

- provides two variants of each C++ class that either store all attributes or only a subset marked as persistent. In many applications, not all attributes are required all the time (residuals in linear algebra, e.g.) and this feature allows the programmer to switch.
- offers support (also via MPI) for complex-valued arithmetics.

> Core statement from [17]: With DaStGen's mapping, we can reduce the memory footprint of the application to the theoretical minimum and send solely relevant information through the computer's network.

Due to the tool and the spacetree's linearisation—a topic discussed next—the present implementation induces a memory footprint for dynamically adaptive grids that is by an order of magnitude lower than comparable libraries or frameworks that rely on a graphs, i.e. pointer data structures [19, 91].

4 Serialisation vs. sequentialisation



The present grid storage and traversal management underlying all discussed papers relies on $2d + 2$ stacks to store all spacetree data. Application-specific properties can be embedded into these stacks or held separately on a heap. From [92].

Our event programming model makes the grid traversal glue the application to the grid data structure and the unknowns. The storage of the latter is subject of discussion in this section. A *tree structure* for dynamically adaptive grids *can be mapped onto a graph* realised with pointers. Because of the structuredness of the spacetree, such a realisation is hardly elegant: memory booking becomes discontinuous, logically/spatially connected data becomes scattered and the pointers introduce an overhead. The term 'becomes' refers to the evolution of memory characteristics as the grid changes. Periodical data reordering can, at additional cost, eliminate scattering. Other techniques such as storing children of one cell en block [45, 46] tackle the memory overhead. To avoid the drawbacks completely, many spacetree codes rely on a total *serialisation* of the tree. Linearisation is an alternative term [85]. We emphasise that a serialisation does not constrain the programming model in any way. Random access still can be allowed. Yet, if the 'randomness' of the data access does not anticipate the data order, the memory access becomes non-local. As the present papers realise a prescribed processing, *the stream can be ordered according to the traversal order and thus data can be read continuously* which is advantageous on current hardware. Serialisation-based spacetree codes supporting dynamic adaptivity require careful design. Insertion of new grid elements as well as deletion have to be available without major data movements.

> Core statement from [91]: Serialisation with unconstrained dynamic adaptivity is possible if the tree traversal reads the tree from a serialised input stream and pipes the grid data afterwards into another stream. The latter can be used as input data next if the traversal direction is inverted after each traversal. The input/output streams then are two stacks. Together with the call stack of the system, three stacks are sufficient to encode the tree structure.

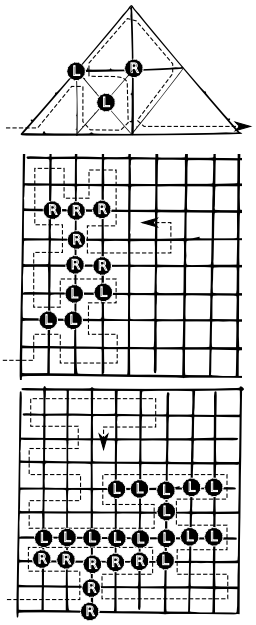
Tree serialisation without embedded data requires one bit per cell to encode whether a node of the spacetree is refined or not. A second bit might be used to control the dynamic refinement. The two bits are often complemented by few further flags simplifying the programming. All papers discussed here rely on the aforementioned DaStGen tool [17] which allows us to *store all the control bits within one char per spacetree node*. This also holds if the bits encoding the spacetree are augmented by information about the traversal order.

Depth-first vs. breadth-first. There are various serialisation alternatives, but the *two basic variants are an ordering along a depth-first order* (denoted as DFS with S referring to the relation to classic search algorithms) *or a breadth-first order* (BFS). Hybrids are possible. Pros and cons run like a red thread through the subsequent sections. The main advantage of DFS is that it allows us to use plain recursive programming and to realise memory modest algorithms with low memory footprint: there are no large temporary data structures/call stacks involved. We simply read in a bit stream. BFS instead requires an additional queue or stack besides the call stack. A serialisation with DFS alone does not yield a spacetree serialisation yet, as the ordering of the children of one node remains undefined [91, 92]. Once we define this order, it can be encoded within a pushback automaton. If the same order pattern is employed on each spacetree level, we end up with the motif of a space-filling curve (SFC) [3, 77]. Historically, most papers introduce the spacetree starting from an SFC-based grid traversal, i.e. argue the other way round.

DFS and BFS impose an order on traversal automaton transitions. If we abstract from DFS and BFS, i.e. try to hide which one is used, we end up with a small set of constraints that hold for both. As example, a child cell a of a cell b is always visited the first time after the automaton has ran through b. *Such simple observations motivate a minimal partial order on all events* in [92]. It is used in Section 5 and 6 to realise parallel applications: If the application code is written such that it relies only on this partial order as invariant, and such that it allows for the parallel invocation of events otherwise, then the traversal automaton may run in parallel.

Space-filling curves. Space-filling curves are a popular tool in scientific computing [3]. They can be classified at hands of a multitude of properties. Of particular interest here is the distinction of connected and non-connected curves. The Morton order/Lebesgue curve is an example for the latter. All implementations here stick to continuous ones where *any two cells visited after each other according to*

the space-filling curve share a common face. For $d = 2$, these curves classify all vertices and faces of a given grid level into vertices/faces left of the curve and right of the curve. Element-wise traversal codes that implement a deterministic, connected tree traversal thus can manage the whole grid administration via stacks. The vertices and faces are read from an input stream. This mirrors the reading of the spacetree description. If they are adjacent to cells read later on, they are temporarily stored on left/right temporary stacks. Once they are not required anymore, they are written to an output stream.



Single-level Sierpinski (top), Hilbert (middle) and Peano (bottom) curve cluster all vertices into left and right. Once they are read in, they can be temporarily stored on only two stacks. **This scheme breaks down if multi-scale data is introduced**. Additional stacks are required then and the access scheme becomes more complicated.

The scheme is straightforward to understand for two dimensions and a planar grid without hierarchy. It has been implemented for the Hilbert curve ($k = 2$), for the Sierpinski curve with bi-partitioning on triangles, and the Peano curve ($k = 3$) [3]. However, it becomes technically challenging for $d \geq 3$ and breaks down if the grid holds data on all levels of the grid rather than only the finest level. Then, *the simple idea with left and right does not work anymore.* We have shown in [91] that, for the Peano curve, one has to introduce additional stacks to overcome multiscale problems and to switch from a left/right classifier to an odd/even classifier along each Cartesian coordinate axis. This scheme ends up with $2d + 2$ stacks in total—in contrast to exponential numbers of stacks used by codes before [50, 55, 73]. While the storage idea with $2d + 2$ stacks stems from [91] and has been and is used by several papers, *formal correctness proofs for all aspects of the grid administration* are introduced with [96] first:

> Core statement from [96]: With the Peano curve and DFS, a vertex or face data management can be implemented with $2d + 2$ stacks even if vertices are to be held on each grid level, i.e. if a vertex is unique due to its spatial position plus its level (cmp. [91]). The proof of this statement results from a folded induction over spacetree depth and spatial dimension d.

Advantages and disadvantages. Three properties of the Peano space-filling curve make the induction step work. They are called *projection*, *palindrome* and *inversion* property. The properties can first be found in [66] with these names. Only the Peano curve seems to have all three properties for arbitrary d. This is a common assumption today—see [58] who also sketches what a proof might look

like. *It also remains doubtful whether a comparably elegant and simple traversal of a serialised spacetree with a fixed number of stacks or heaps could be realised with a BFS.* A small, fixed number of stacks or heaps is essential to end up with an algorithm that has advantageous memory access characteristics, i.e. whose memory accesses exhibit high temporal and spatial locality [65]. *Such a scheme is memory hierarchy- and cache-oblivious.*

DFS is the starting point for any algorithm here, while we emphasise that BFS is a 'better' alternative from an HPC point of view as it allows us to employ (loop-based) parallelism straightforwardly. DFS serialising the spacetree and using stacks is strictly sequential. Therefore, significant effort is spent in the papers discussed from hereon to weaken the DFS' sequentiality without changing its fundamental memory access characteristics. *Besides the DFS' sequentiality, studies* of the results of [76, 87, 93] and related papers reveal that *the stack-based realisation also suffers from a very high integer arithmetics overhead.* All the recursive, localised traversal routines are elegant from an algorithmic point of view. However, they face the, typically small, computational (floating point) work per vertex or cell with high administrative work. Tackling this second challenge is of particular interest in [76] and [97] where high vectorisation efficiency is an objective. It is also briefly picked up in [92] where we furthermore give details about typical programming workflows and development tools we have created to simplfy the work with our spacetree code.

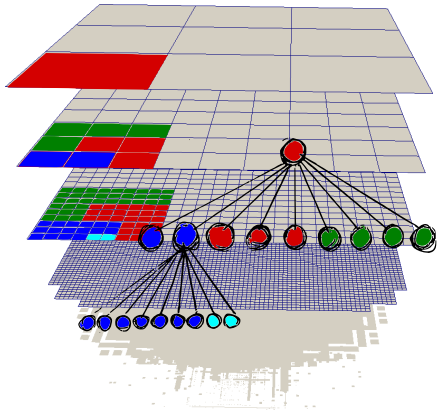
5 Tree decomposition on distributed memory

Serialisation with an SFC renders a plain parallelisation of the traversal impossible. The traversal's concurrency level is one. Within a distributed memory environment, we thus decompose the spacetree and make multiple automata process their own tree.

Decomposition variants. In [91], we introduce for this purpose a decomposition of the tree where subtrees are cut out of a global tree and deployed to ranks of their own. A quick comparison to other papers (cmp. [92]) reveals that there are basically two competing decompositions with the tree cut decomposition being the less popular one (though [1] claims that it is the natural choice for multiplicative multigrid or octrees in general—a statement that can be doubted given the vast number of approaches following the competing strategy [2, 22, 23, 49, 54, 60, 67, 74, 79, 80, 84, 85]): We can either start from the root and recursively deploy cells together with all their descendants (children, children of children, ...) to other ranks, i.e. we cut out trees from the global spacetree and deploy these trees to ranks. Such *a top-down approach uniquely assigns each cell on each level within the spacetree to one selected rank.* It defines a non-overlapping cell decomposition on the finest grid. And it is non-overlapping in terms of the tree. Or we can decompose the finest grid level into subdomains and deploy these domains to ranks. A refined cell within the spacetree then is held by a rank if this rank holds at least one of its children. Such *a bottom-up approach starts from a non-overlapping*

cell decomposition on the finest grid, too, but holds some refined cells redundantly. Eventually, the root node is held on all ranks. It is overlapping in terms of the tree. The two approaches are sometimes also classified by the terms horizontal (bottom-up) and vertical (top-down) splitting with 'vertical' referring to the grid levels and 'horizontal' naming splits within grid levels or the adaptive fine grid.

Programming model. A programming model for both variants arises naturally: For the bottom-up decomposition, the root of the global spacetree is held redundantly on each rank. Each rank starts its local spacetree traversal in parallel. For a top-down decomposition, we basically follow the same approach, though different automata start on different tree levels, i.e. with different mesh results. *Differences arise once an algorithm requires vertical data flow*, i.e. information transport from fine to coarse grids or the other way round [92]. Otherwise, both schemes are similar whereas the bottom-up approach avoids redundant data storage. The data exchange via boundaries—typically the vertices and faces along subdomain boundaries are held redundantly and data is exchanged 'through' these grid entities—is the same. We detail it in Section 7.



Top-down splitting cutting out whole trees from the spacetree recursively. Each cell on each level is uniquely assigned to a rank. The tree splits induce a tree topology on the ranks. From [92].

Vertical data flow in the bottom-up approach remains a local operation as each rank holds all coarser levels. However, we might have to fuse dispersing trees on coarse levels, i.e. keep the multiple coarse level copies consistent with each other. The coarser a cell's level the more ranks hold copies of the cell. That is the underlying *communication graph becomes the denser the coarser the grid level.* Vertical data flow in the top-down approach requires data exchange via message passing if the flow bypasses a tree cut. Data consistency is ensured by construction since each cell has a unique 'owner'. However, *as the data flow integrates into the automata traversal, it is latency-sensitive* [97] *and partially synchronised*: Automata might not be able to start 'their' traversal before the rank responsible for coarser levels has run through specific cells. Automata might not be able to unroll the recursion stack before some remote automata responsible for finer trees have finished their traversal. This observation results directly from the fact that the recursive tree splits induce a logical tree topology on the MPI ranks with masters and workers. Data flow from the master to the worker requires that the master's traversal already has entered the parent cell of the worker's local root. For restrictions, i.e. data transfer from workers to masters, the master's traversal has to wait for the worker respectively.

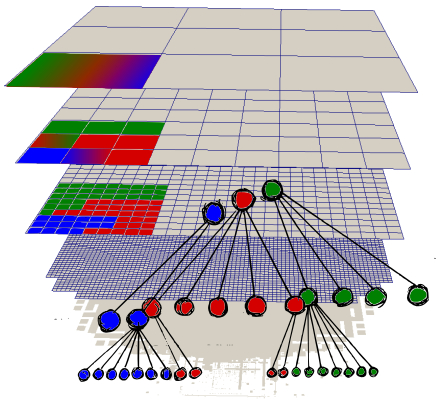
We finally emphasise that a bottom-up splitting poses no constraints on fine grid partitions. A top-down splitting makes all subdomains logically map onto a whole spacetree. *Not every fine grid decomposition is admissible.* While experiments studying small rank counts (see results presented in [91]) have to tackle this problem

by MPI overbooking, e.g., studies with bigger rank counts do not report to suffer from this property. In practice, multiple MPI ranks are ran per node anyway, and other communication properties hide small load imbalances induced by the admissibility constraints.

> Core statement from [92]: The tree splits (top-down decomposition) avoid redundant data storage (and computation) as found in classic (SFC-based) spacetree parallelisation schemes that start from a fine grid decomposition. This advantage comes at the cost of a tighter traversal coupling and constraints on admissible fine grid partitioning.

A parallel execution of particular events is made possible due to the fact that the programming model imposes solely a partial order. We add that the MPI variant of the code augments these events by additional events that allow users to plug into boundary, top-down and bottom-up data transfer via MPI. As each rank runs an automaton triggering the same set of events, the parallel programming paradigm thus follows *Single Program Multiple Data* (SPMD).

Level-wise DFS and rank synchronisation. Our event-based programming model allows us to hide the technical realisation of the tree splits. The latter rely on two ingredients: on a one-level recursion unrolling and on the partial order imposed on the events. One-level recursion unrolling is introduced in [91] as *level-wise DFS*, and it acts as blueprint for more aggressive unrolling as discussed in [31]. In the context of tree decompositions, it means that all cells and vertices that are direct children of a refined cell are loaded before the automaton descends into any of them. Per level, it is a hybrid of depth-first and breadth-first traversal. refined cell, trigger remote traversal on the children that induce a remote spacetree, continue with the local subtree, and to wait afterwards for all remote traversals. This way, *level-wise DFS preserves the partial orders for vertical data flow* as discussed in Section 6 and formalised in [92].



Bottom-up decomposition starts from a fine grid splitting. A coarse cell is assigned to a rank if at least one child is held by this rank. This induces redundant coarse grid storage. From [92].

Latency-sensitive realisations become problematic on today's hardware. The particle studies in [97] show measurements where the hardware's logical network tree can directly be identified at hands of scaling graphs. Whenever upscaling of the problem requires additional nodes connected through an additional switch, the code faces a sudden performance break-down. *The data transfer top-down and bottom-up per global tree traversal suffers from latency.* Motivated from this insight, we state:

> Core statement from [97]: Partial skips of vertical data exchange asynchronises the parallel traversal automata. The resulting higher concurrency level makes the traversals less sensitive to latency effects.

We therefore propose for our code base to offer the opportunity to the application-specific codes to control the vertical synchronisation between automata with high granularity.

> Core statement from [92]: It is advantageous to grant applications control globally or on a rank-to-rank base whether data are sent top-down or bottom-up. This allows the underlying traversals to skip the synchronisation between ranks and to increase the asynchronity.

6 Recursion unrolling

Recursion unrolling has been introduced in the SFC-automata context in [91] as well as in the previous section in the context of level-wise DFS. It gains significance in [31] and [81] where it serves for a different purpose: *Recursion unrolling helps to reduce the traversal's high integer overhead and allows us to exploit manycore architectures due to bulk synchronous processing* (BSP). It weakens the strict sequentialisation due to the serialisation, and it speeds up the traversals significantly in grid regions that represent regular subdomains on one rank.

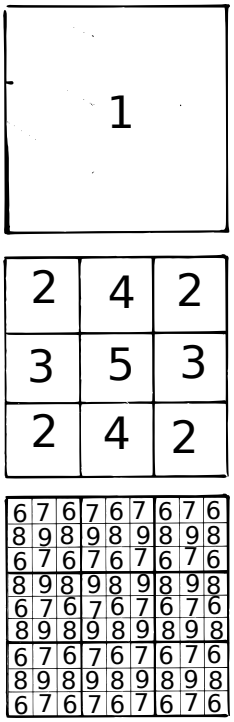
Analysed tree grammars. We propose to pick up Knuth's classic definition of analysed tree grammars [64] and to assign each stationary leaf, i.e. each leaf where we know that it does not change in the subsequent traversal, the marker 0. All other leaves obtain a marker $\perp$. Throughout the DFS traversal, we analyse the marker bottom-up when the traversal backtracks, i.e. unrolls the call stack and, thus, ascends in the tree. Any refined cell is assigned the marker c if all children have been assigned the marker $c-1$. Others get $\perp$. We furthermore may invalidate any marker to $\perp$ because of dynamic refinement or coarsening, received MPI data or dynamic load balancing. We end up with a scheme *that identifies subtrees that yield a regular subgrid and will remain invariant in the subsequent traversal.*

If the traversal automaton descends into a refined cell flagged with a marker c, it knows that this cell induces a full spacetree of depth c. It is thus straightforward to allocate memory for such a tree temporarily and to permute the steps on it: the automaton first loads all cells and vertices, it then invokes all the events, and then stores all data again. Preserving the partial order on events from [92], it is furthermore natural to resort the events. Most obvious, we may change from a depth-first order into a breadth-first order. For the latter, we may use concurrent event calls on every level.

> Core statement from [31]: An analysed tree grammar tracking invariant regular subtrees gives us the opportunity to switch within these subtrees to a concurrent breadth-first ordering following standard colouring schemes. Formally, such an approach equals recursion unrolling.

While BSP parallelisation is a feature resulting from recursion unrolling that is reported in [31], another improvement of the approach is the *elimination of the recursion overhead*. It yields faster code. This effect is described in [81] where we use the grammar for further optimisations. Prior to a discussion of these optimisations as well as the main observations from [81], we return to shared memory parallelisation.

Inter-cell concurrency. In [31], we propose to use 2^d colouring on regular subtrees: once the whole tree is loaded, run through the levels one by one. This is a BFS. Within each level, invoke all events concurrently whose cells do not share data. This is an extension of red-black Gauss-Seidel colouring for 3^d data dependency stencils on the cells. Following data dependency considerations on cells, we may also introduce colourings with more colours that automatically ensure that two-level operators are evaluated on a shared memory environment without any data races. 3^d colouring for cells or 5^d colouring for vertices yield appropriate orderings for most operations though it depends on the application's data access needs. Such a strategy is used, for example, for geometric-algebraic multigrid based upon BoxMG in [90]. Following [92], it is advantageous to allow the application to define per event which concurrency level is feasible. Concurrent execution can be switched off completely—required by plot routines, e.g.—all cells and vertices can be handled concurrently, or any colouring can be chosen per algorithm phase. *The user code formulates what is done within events and prescribes constraints on these events.* How events are invoked in parallel on regular subdomains is left to the computational kernel.



Recursion unrolling for a subtree with $c = 2$ loads all three grid levels into three regular Cartesian grids (temporary buffers). The depicted cell traversal order results from 2^d-colouring.

The shared memory parallelisation introduced with the recursion unrolling is classified in [93] as *inter-block concurrency*. There, it refers to blocks embedded into spacetree leaves. Abstracting from a particular application domain, it should be called *inter-cell* concurrency. It differs from *intra-cell* concurrency where kernels acting on one cell or vertex, respectively, are parallelised internally; a strategy that has nothing to do with the spacetree. Besides these two paradigms, many functions

in our spacetree traversal code base are multithreaded as well. However, the major speedup stems from the parallel treatment of cells and application codes that parallelise activities performed within the cells. The two parallelisation schemes are technically orthogonal.

k-spacetrees. In the context of patches embedded into cells and the general notion of a k-spacetree, it is important to emphasise that recursion unrolling can be rewritten as temporary gluing of patches or a temporary increase of k. If a subtree is unrolled, the unrolling is equivalent to replacing the tree by one coarse grid cell that hosts a sequence of regular grids. If a subtree is unrolled and if algorithms focus solely on the finest grid level, the k in k-spacetree is formally changed into k^{c+1} and the whole subtree is replaced by a k^{c+1}-spacetree of height one. An alternative formulation of a code embedding patches of x^d cells into the leaves is to use a k-spacetree where each refined cell holds a marker from $\{\perp, x, x+1, x+2, \ldots\}$. This assumes that x is chosen as multiple of k. These observations are not documented [31, 91, 93, 97].

> Core statement from [93]: The idea of recursion unrolling due to an analysed tree grammar applies to patch-based formalisms as well and shows how we may replace assemblies of patches in regular grid subdomains by larger patches embedded into coarser levels of the spacetree.

7 Partition boundaries and regular subtrees

For the final methodological section, we return to the DFS-SFC combination. Since this team induces a total order on the whole spacetree, it also induces a total ordering on any subtree. Since it serialises any subtree, *it also defines two orders on all vertices on the multiscale boundary of its subdomain: one for the first run-through through any adjacent cell and one for the last possible write by an adjacent cell.* Section 4 relies on a stack-property of the Peano SFC to show that all persistent data can be held on an input plus and an output stack. As we fall into SPMD for the MPI programming, each local traversal automaton follows the same SFC direction per traversal and all automata synchronise their switch of the traversal direction after each multiscale grid sweep.

> Core statement from [81]: Data along partition boundaries can be exchanged between ranks without any reordering, if all data sent away in one sweep is received and merged into local data structures in the subsequent traversal.

MPI data exchange. Such a communication pattern resembles the Jacobi unknown update in linear algebra, as changes to a vertex along the domain boundary

are not available to other ranks prior to the subsequent sweep. The ordering simplifies the MPI data exchange since it realises, per communication partner, one plain stream without any sorting or reordering. The streams can, depending on the hardware characteristics, be buffered locally and submitted in chunks. The idea was realised in very early Peano-SFC-based implementations [60, 67] and formalised in [91]. The generalisation to other SFCs can be found in [81]. It is worth a note that the SFCs' Hölder continuity yields direct estimates on how big temporary stacks can grow for regular grids. We footnote that, to the best of our knowledge, a formal estimate for adaptive grids is not available yet. *All estimates on the locality of the SFCs imply that the data exchanged via MPI is small compared to the volume held in SFC-induced subsections.* Observations on this quasi-optimal surface-to-volume ratio can be found in [49] for Hilbert indices (the paper also lists related literature), and we generalise it to the Peano SFC in [20]. Again, we highlight that we do not only exchange a small number of vertices per rank. We exchange only those attributes of the application's data model that are explicitly required. The generation of the user-defined MPI data types that are required to realise such a subset exchange is again outsourced to DaStGen [17].

Block memory access. In [81], we distinguish between a 'shared memory' decomposition and a 'replicating data' layout for shared memory and distributed environments. The replicating data decomposition equals the aforementioned tree decomposition scheme for MPI. Obviously, it also can be ran on shared memory systems. The shared data decomposition splits the domain into chunks referred to as clusters [80] but makes vertices on the partition boundaries not stored redundantly. Instead, traversal automata hold meta data to compute where they are stored. Subject of discussion in [81] is a *compression of this meta data along the SFC order*. It is run-length encoding (RLE). In practice, the impact of such a compression on the whole code is limited as the meta data memory footprint is small compared to the actual data.

The impact of meta data RLE compression—which basically is reduction in a multiscale sense in [81]—is more significant in terms of runtime for the shared memory parallelisation once we fuse it with recursion unrolling. Meta data per cell can encode how many vertices are to be read from the input stream and how many vertices are written to the output stream due to the particular cell. Obviously, such data can again be analysed bottom-up along the k-spacetree. It results from a reduce with an addition operator. Throughout the subsequent traversal, not only can we pipe all input data into a set of Cartesian grids; we also know through the meta data how many vertex and cell records have to be read. Statements on the writes follow analogously.

> Core statement from [81]: A bottom-up analysis of how many vertices are read by cells for the first time or written for the last time allows us to load continuous chunks of vertices from the input stream for regular subtrees or to store continuous chunks of vertices respectively.

Obviously, reading data in chunks is superior to the step-by-step reads induced by a recursive traversal implementation. Furthermore, the option arises to split the chunk to be read into smaller chunks which are read concurrently by multiple threads. If a regular subtree has to read x vertices, a first thread may read only $\hat{x} < x$ vertices. We spawn this thread, shift the pointer along the input stack by $\hat{x}$ and continue. This approach *allows us to exploit multiple memory controllers due to multiple read threads within regular grid regions.* It is the major runtime improvement reported in [81] that is made possible by analysed meta data.

8 Results from the case studies

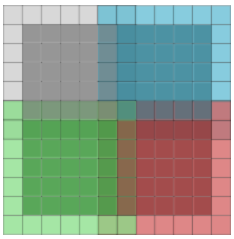
Sections 3 through 7 present new methodologies in the spacetree-SFC context. The methodologies' core statements propose answers to the implementation questions arising from the case studies. In this section, we reiterate which algorithmic ingredients enable the papers to achieve their break-throughs.

A shallow water simulation. The SWE use case embeds patches into the spacetree. As the spacetree automaton traverses the grid, it runs through the set of patches and can invoke the kernels operating on these data structures. The spacetree then acts as organisational data structure on top of the actual compute data. With the heap provided, we can make each cell/leaf of the spacetree point to its patch, i.e. its compute data. Furthermore, we can add information to the vertices which compute data 'is adjacent' to each vertex. Since the traversal automaton entering a cell has access to all its vertices, it consequently also has access to all neighbour patches and can initialise the ghost cells.

The alternative data storage, i.e. an embedding of the patch data into the data stream, would be problematic, as the streams then grew large. Since the streams are always piped from an input into an output stream to facilitate dynamic adaptivity, holding organisational and compute data separately is advantageous—in particular if we support local time stepping where not all patches are updated in each traversal [87]. The impact of DaStGen and other memory footprint reduction techniques are not dominant in the SWE code. The memory footprint is determined by the actual patch data together with the halo data. However, we retain excellent cache behaviour as long as all patches are chosen such that they fit into the caches, as patches continue to be processed in a localised fashion and ghost data exchange thus exhibits spatial and temporal memory access characteristics, too.

For a patch-based code whose kernels do not exploit multiscale, all data access is horizontal data exchange. The SWE solver's explicit time stepping updates patches starting from their 'old' solution into a new one. A Jacobi-style unknown update—new solutions are sent after a patch update and received prior to the next update of neighbouring patches—thus makes latency considerations and vertical data flow studies irrelevant. We have to expect the code to suffer from limited network bandwidth, though MPI parallelisation is not subject of discussion in [93].

Recursion unrolling for shared memory however is used for the organisational data structure and allows the code to apply a 2^d-colouring on the patches. Patches of the same colour, i.e. whose ghost layers do not overlap, then are updated in parallel. Our results reveal that the inter-cell concurrency of the spacetree with recursion unrolling and the intra-cell concurrency introduced by the kernels working on patches are conceptional orthogonal, but do interfere. Inter-cell parallelisation pays off if a grid hosts large regular subdomains. Intra-cell parallelisation pays off if the work per grid entity is large. Any application of either scheme changes the memory access characteristics and the computational load, i.e. in terms of performance the two approaches are not orthogonal.



Four patches are fused into one patch logically assigned to a coarser spacetree cell. We eliminate ghost layer exchanges, can provide kernels processing regular Cartesian grids with higher vectorisation efficiency, and apply simple for-loop parallelisation (intra-cell concurrency). For $d = 2, k = 3$, at least nine patches have to be used. The patch assembly is broken up if adaptivity criteria require the kernel to do so. From [93].

The SWE studies are the prototype of a code where the analysed tree grammar is used not only within the tree. We also use it to fuse small patches in regular regions into one big patch. Efficient stream-like data exchange along subdomain boundaries again is not important here as the majority of data to be exchanged is halo/ghost data which consists of regular chunks associated to the patches anyway. An efficient realisation of load and store operations within the spacetree plays a minor role as the majority of runtime is spent within the patches, i.e. the ratio of patch work to traversal work is high. *Among the technical details, it is primarily the interplay of heap-based patch fusion, analysed tree grammars, recursion unrolling and patch fusion that allows us to use the Xeon Phi.* Without the fused patches, we would not be able to exploit the wide vector registers and the massive core count that is well-suited for in-order parallel-for. Without the dynamic decomposition of fused patches into their constituents again, we would not be able to offer unconstrained dynamic adaptivity. We summarise:

1. **Patches and arbitrarily dynamical adaptive grids** can be brought together within the spacetree efficiently if cells host the data on the heap and adjacency information is stored within the vertices.

2./3. We may start with very fine regular patches and on-the-fly fuse assemblies of these patches into bigger patches. For fused patches, overlaps due to ghost cells are lower, we can write code that vectorises efficiently, and we can rely on simple parallel-for parallelism within the patch. All this works without any constraint on the dynamic adaptivity and **without severe constraints on minimum patch sizes**, though constraints make sense from a performance point of view. The dynamic fusion furthermore allows us to **balance between good vectorisation properties and reasonable inter-patch concurrency**.

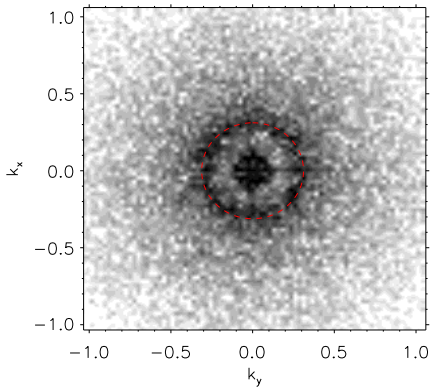
While the experiments in [93] reveal that we can obtain excellent throughput, it remains an open question whether and how one should design corresponding adaptivity criteria that push the grid towards regularly tessellated subdomains that can be fused into high-efficiency coarse scale patches. The idea to realise densified, highly efficient linear algebra kernels tailored to specific block sizes [59] integrates seamlessly into the proposed scheme. It closes the performance gap to codes which rely on specialised kernels for specific grid regions (see for example [43] and references therein) but are otherwise block-structured, i.e. do not support arbitrary, dynamic refinement. While the present application example realises a benchmark setup, a rather basic numerical scheme and lacks a complex physical model compared to other work referred in this overview [2, 5, 7, 14, 59, 68], it is, to the best of our knowledge, *outstanding in combining unconstrained dynamic adaptivity with high manycore efficiency*. It provides an efficient algorithm due to the spacetree's structuredness but preserves all flexibility. No other code in the field (such as AMR-Claw [25], Flash [30] based upon Chombo or Paramesh, ForestClaw [21], RAMSES [86], or SeiSol [63]) offers this.

A particle-in-cell code. With heaps, we can associate particles to cells. In [97], we however propose a Particle-in-Dual-Tree (PiDT) scheme where we associate particles to vertices. This turns out to be faster and simplify the coding of particle-grid and particle-particle interaction. The latter is beyond scope in this application. If particles move from one cell into another, this is realised as reassignment to another adjacent vertex of the particular cell. If particles move more than one cell—they tunnel—this is realised by a reassignment to a coarser level. We call this *lift*. The lift is followed in the next DFS traversal by a *drop* into the right grid cell. The spacetree acts as organisational data structure for the particles while we propose to realise the PDE solver at hands of the spacetree mesh. Particles contribute with a Dirac distribution to the PDE's right-hand side. As they are stored next to their closest vertex, the evaluation of this right-hand side is straightforward. With the whole picture, i.e. particle formalism plus PDE solve, the spacetree acts in both roles: as computational and as organisational data structure.

Tunnelling is the major difference to other particle-based codes. It is enabled by lifts and drops. They are made possible by to the spacetree's multiscale nature which in addition provides us with an AMR grid. The latter is important as the coupled PDE demands for fine resolution around particle clusters but yields smooth solutions in the remainder of the domain. Particle-based formalisms depend on the opportunity to simulate as many particles as possible, and DaStGen's bit packing reduces the memory footprint of the particles to an absolute minimum. The memory footprint of the grid, in contrast, plays a minor role. The stack-based grid access scheme also makes no major impact, but the DFS-SFC traversal's locality implies that the particle resorting, i.e. the reassignment to new vertices either on the same level or coarser and finer levels, is a localised memory move.

As particles are allowed to travel through the grid arbitrarily fast, each time step of the simulation comprises a global particle sort. Each rank has to notify each other rank whether particles leave its subdomain. The top-down de-

composition of the spacetree fits to this scheme: whenever a particle has to be lifted, it is lifted solely on one rank or is transferred from one rank to another. We show in [97] that those particles moving of at most one cell per time step can in contrast be realised more efficiently as horizontal data exchanges through subdomain boundaries. We benefit from the fact that no data reordering for boundary data exchange is necessary due to the SFC. However, we also show that the code becomes latency-sensitive—a property that seems to be tight inherently to the global resort per time step. Our work introduces a bottom-up analysed velocity check and thus is able to predict whether particles in a given subdomain, i.e. for certain subtrees, may tunnel in the present time step or not. If they can not tunnel as they are too slow, the global reduction within the spacetree is locally switched off. We call this *reduction-avoiding PiDT* (raPiDT).



Electric field spectrum in the wavevector-wavevector plane. From [97].

Among the technical details, it is primarily the interplay of heaps, multiscale domain representation, SFC-based boundary data exchange of particles and the feature to switch off worker-master communication between MPI ranks locally that allows us to make the hard problem of global particle sorting scale. Without the opportunity to switch off reduction, we run into inverse weak scaling: The more particles enter the system, the more particles may tunnel and have to be transferred from workers to masters and the other way round. This makes the global particle re-sort spend all time in particle exchange. The other algorithmic steps (particle-grid projection, PDE solve, solver-particle projection) are state-of-the-art challenges in the field. We summarise:

4. The paper shows that, **because of the heap, particles either can be stored within the containing spacetree cell or can be associated to the nearest vertex**.

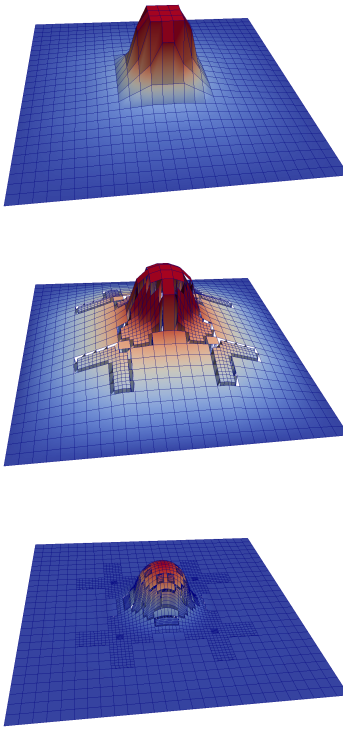
5. A storage within the nearest vertex has advantages.The **particle associativity can be updated incrementally and assignment updates along subdomain boundaries can be realised via particle exchange along the space-filling curve**.

6. While the experiments exhibit excellent scaling for high particle counts and small MPI rank counts (up to around 1.000), they also reveal that the code performance starts to decrease if we increase the particle count further or increase the core count. More time is spent in global all-to-all communication. These measurements holds for the particle resorting. With other solver steps, notably the PDE solve, included in timings, we postpone the critical threshold around 1.000 ranks. However, it continues to exist. The paper introduces an

algorithm that **predicts where data exchange can be omitted locally** as no particles would be sent anyway. Such a **communication-avoiding helps to improve the scaling**.

If tunnelling has to be enabled, we show that we obtain significantly better performance than competing SFC-based schemes (in particular other bottom-up decomposition codes such as [54, 74]) and close the performance gap to standard linked-cell approaches without tunnelling as they are used in molecular dynamics, e.g. While the present application realises a benchmark setup, a rather basic numerical scheme and lacks a complex physical model compared to other related codes relying on a particle formalism (such as Enzo [33], Gadget and follow-up codes [83], SWIFT [44]), it is, to the best of our knowledge, *outstanding in combining reduction-avoiding techniques, a multiscale representation/storage of particles and unconstrained particle velocities*. It provides an efficient algorithm due to the spacetree's multiscale nature and structuredness.

A multigrid Helmholtz solver. Among the spacetree's selling points are its support for arbitrary dynamic adaptivity, in-situ mesh generation, dimension-generic programming, and the inherently built-in multiscale representation due to ragged Cartesian grids embedded into each other. These ingredients make it a well-suited base for p-dimensional geometric multigrid solvers that realise F-cycles, i.e. unfold the grid on-the-fly. While the element-wise DFS grid traversal is well-suited for realise matrix-free stencil operations, it turns out that the additive schemes' data flow does not allow us to fuse a solver iteration and full approximation storage (FAS): Coarse grid update's prolongation makes information flow from coarse to fine grids, whereas FAS' injection pushes information from the fine to the coarse grid. FAS is advantageous to realise dynamic adaptivity with hanging nodes without complex, adaptivity-dependent stencils. This insight is well-known in the multigrid community [15, 16]. We show that pipelining with the introduction of additional helper variables on the grid [38–42] allows us to overcome the data flow restriction and have a tree traversal code that realised one additive FAS cycle per tree traversal.



Solution development of the additive multigrid solver for a Helmholtz problem. From [76].

Since we embed all unknowns as well as temporary records such as residuals into the spacetree stream, the tree acts as computational data structure. DaStGen's reduction

of the memory footprint thus plays a major role—in particular as soon as we run the simulation on Xeon Phis with their limited main memory. Furthermore, the grid management with its excellent memory access characteristics allows us to fuse multiple PDE solves on one grid and compute solutions concurrently. Hereby, we also evaluate coupling terms A_{ij} immediately and matrix-free if equations i and j are fused. Recursion unrolling of whole regular subtrees speeds up the code. However, we rely on an F-cycle, i.e. the solver permanently adds additional grid levels. This limits the tuning's impact as regular subtrees have to be broken up with new vertices entering the grid.

Of particular interest in [76] is the fact that all multigrid implementations techniques—in particular regarding the pipelining—are proven to be correct. *Among the implementation details, the interplay of DaStGen's efficient data modelling, the stacks, the multiscale nature and the dimension-generic formulation of the tree allow us realise a robust and efficient multigrid code.* With weaker memory access characteristics, the fusion of multiple PDEs within one grid would not have led into a code with high vectorisation efficiency. With only a slightly higher memory footprint, we would have been limited to even smaller problems on one card. This is critical, as the problems are in principle p-dimensional with p being the particle count. Currently, $p \in \{2, 3\}$ are state-of-the-art. We summarise:

7. DaStGen's data generation **supports complex-valued data models and arithmetics**. We thus can use all algorithms straightforwardly in a complex-valued setting.

8. Pipelining or related techniques that require codes to **embed additional helper data into grid entities do not make the performance deteriorate**. DaStGen's C++ augmentation render the modification of the data model straightforward.

9. If we solve multiple coupled PDEs on one grid and if it is possible to **integrate the equations' coupling into the solve**, i.e. each iteration step immediately affects the other solves), this increases the **arithmetic intensity superlinearly** and allows us to end up with high vectorisation efficiency.

More sophisticated ideas proposing problem-specific intergrid transfer operators within the spacetree world [90, 98] or stronger smoothers [39, 41, 42] integrate seamlessly into the proposed scheme. It will close the gap to established multigrid packages. The idea to realise matrix-free multigrid solvers within a SFC-spacetree environment is mainstream and old [1, 34, 49–51, 55, 66, 71, 73, 79, 84, 85, 91]—in particular as the spacetree yields a hierarchical generating system which simplifies the programming [47, 48]. However, the number of complex-valued multigrid solvers that are available freely is limited, and few codes support grids of arbitrary spatial dimension. While the present application example realises a benchmark setup, a rather basic numerical scheme (low-order discretisation and Jacobi-based smoothers) and also lacks a complex physical model it is, to the best of our knowledge, *outstanding in combining unconstrained dynamic adaptivity with pipelining,*

FAS and matrix-free computations which helps to deliver high arithmetic intensity at low memory footprint and a robust solver for the Helmholtz problem for arbitrary dimension. It is able to provide this melange of features due to the spacetree's structuredness.

While we would like to add a statement 'for arbitrary particle counts p, i.e. for p-dimensional grids', such a statement would be wrong: while the infrastructure supports any p, we find the solver's smoothing properties already deteriorate for $p = 4$. Smoother issues are well-known for classic bipartitioning. Peano's tripartitioning makes this challenge harder. Although we obtain constant cost per degree of freedom, the number of degrees of freedom explodes for large p—the well-known curse of dimensions—and thus makes solves for large p still unfeasible and demands for alternative approaches [18].

9 Conclusion

Résumé. There are two options to read the present document: It can be either read as a collection of algorithmic ingredients useful in the spacetree context, or it can be read as high-level digest of a software ensemble that integrates the presented ingredients into one code base. This code base is called Peano [94]. The name pays tributes to the underlying SFC. Both interpretation variants lead to different conclusions. *The ideas also can become of value for any other spacetree-based coding projects and can help to push other codes forward.* They cover the topics multiscale data modelling with low memory footprint, automaton-based programming paradigms, grid-data associativity, stream-based and memory efficient data management, flavours of tree decompositions well-suited for distributed memory environments, recursion unrolling techniques for lightweight shared memory parallelisation, and efficient data exchange between regular subtrees in a grid as well as distributed subtrees. Furthermore, three contributions to particular application areas are made. They cover patch-based manycore exploitation, communication-avoiding particle resorting and FAS-based additive multigrid codes. *As a code base, this code base can be used for rapid prototyping of dynamically adaptive multiscale solvers running on parallel machines.* In the latter case, the unique selling point of the code base [94] is the integration of many sophisticated ideas while other codes might be several steps ahead at hands of any particular metric.

Classification of spacetree codes. Numerous spacetree codes do exist. Some of them claim to be general-purpose, others were written with a particular application in mind. To give a comprehensive overview of the different software variants is out of scope for the present text. However, it is possible to use the presented core concepts to introduce some classifiers differentiating the present work to other approaches. This scheme is not comprehensive, but it facilitates a comparison.

- **Regularity.** Adaptive Cartesian grids have to feature hanging nodes. We have to quantify this adaptivity. Some codes constrain the adaptivity and permit two

neighbouring leaves to differ at most one in their level. In accordance with general meshing terminology, such a constraint yields k-1-irregular meshes or is called k:1 balancing [79]. We do not enforce such a constraint here though realising it on top of the present code base is possible. While we thus do not run into the rippling effect [79], the number of hanging vertex configurations in principle is not bounded. To avoid the coding of complex, general-purpose code fragments tackling all possible hanging node configurations and to avoid an embedding of triangular meshes along resolution boundaries [8]—both approaches increase the application source code complexity—we propose to use the MLAT idea and hierarchical generating systems [15, 16, 47, 48] as in [76].

- **Dimensions.** Most spacetree codes are written for two-dimensional or three-dimensional settings. Few support both $d = 2$ and $d = 3$. As the spacetree concept is dimension-generic, spacetree codes in principle also can support any $d \geq 2$. However, to the best of our knowledge, only the prototypical software coming along with [55] provides arbitrary d. The limitations on d regarding the curse of dimension is highlighted in [76].
- **Persistency.** Not every application requires the software to hold the spacetree in memory or out-of-core persistently. It can be sufficient to reconstruct the spacetree on-the-fly for particular algorithmic phases. Notably for particle-based formalisms reconstruction seems to be practical [74, 85]. The present work focuses on a persistent representation of the spacetree in memory.
- **Extent.** A fundamental design question for any author of a spacetree code is to decide whether the whole tree is to be stored/administrated or whether it is sufficient to focus on the finest level of the tree only (cmp. [8, 23]). The latter mirrors a restriction on Ω_h, whereas the whole tree provides a geometric multiscale hierarchy. Such a code holds all $\Omega_{h,\ell}$ simultaneously [6, 76, 92, 96]. We follow the latter option.
- **Role.** Spacetrees offer efficient ways to identify adjacency information and multiscale relations. Connectivity and topology are key ingredients of any mesh administration [4, 8, 14, 23, 49, 79, 85] and hence legitimate the use of spacetrees. However, it is also possible to make the spacetree hold the application-specific unknowns as well. We can distinguish between two different realisation variants for the latter that pick up our storage discussion: either the spacetree yields a direct index to access data stored separately in containers such as hash maps [49, 87] or the spacetree data comprises application data, too. We propose to support both variants and to choose for each solver the better-suited one [93, 97].
- **Linearised data and data access.** It is important for the realisation of algorithms on top of spacetrees whether data—either embedded into a linearised spacetree or indexed by the spacetree—can be accessed in arbitrary order or whether the order is fixed. This is a fundamental distinction when we discuss algorithm realisation variants and methods to couple spacetree codes with existing solvers. The present papers focus on a deterministic, prescribed traversal order as well as linearised spacetrees.
- **Geometric extension.** The present work sticks to a pure spacetree formalism with an invariant number of children per refined tree node, though we highlight

the interplay of the k-partitioning with recursion unrolling. This leads to poor boundary resolution of $\mathcal{O}(h)$. Other approaches relax this constraint and switch to an unstructured coarse mesh hosting a forest of octrees [23, 43, 84] or introduce at least anisotropic refinement [45, 46].

Future work. The present overview illustrates that spacetrees have become very mature. While some features deserve additional attention—notably the differences between top-down and bottom-up decomposition have to be understood better and hybrids have to become available— it seems that most improvements in the data structure context today are evolutionary rather than transformative.

Associated areas of research seem not to be understood completely: in particular, load balancing for heterogeneous hardware with fluctuating performance as well as lightweight, resiliency-aware on-the-fly decomposition of data pose timely and urgent challenges that have to be tackled now with the dawn of the exascale era and the arrival of manycores. Furthermore, the three chosen use cases show that there is a vast set of open research questions on the application side. In the hyperbolic context, local time stepping seems to become the number one challenge with severe implications for the load balancing. Communication-avoiding and communication-aware data exchange gain importance due to the deep, heterogeneous communication topologies. Robust and scaling multigrid ingredients are evergreens of linear algebra. For other application domains, other challenges may arise.

Though the grid management and administration is mature, one fundamental challenge remains, in my opinion, completely open: future applications will require high-dimensional problem solves. The Helmholtz solver [76] here is an example. Space-time approaches yield further use cases [52, 56, 57, 61, 70, 82, 95]. Yet, in particular optimisation, calibration and uncertainty setups that rely on adjoint formalisms and span the parameter space will require high dimensional grids. Without a fall back to multishoot/Monte-Carlo methods studying ensembles of simulation runs, only sparse grids [18] so far promise to overcome some of the problems with the explosion of unknowns. It will be interesting to see whether classic spacetree codes can evolve into high-dimensional data structures as well—probably due to an anticipation of sparse grid or stochastic sampling ideas.

References

[1] M. F. Adams, J. Brown, M. Knepley, and R. Samtaney. Segmental refinement: A multigrid technique for data locality. Technical report, 2015. arXiv:1406.7808 [math.NA].

[2] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 52–, New York, NY, USA, 2003. ACM.

[3] M. Bader. *Space-Filling Curves—An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering*. Springer-Verlag, 2013.

[4] M. Bader, C. Böck, J. Schwaiger, and C. A. Vigh. Dynamically adaptive simulations with minimal memory requirement - solving the shallow water equations using sierpinski curves. *SIAM Journal of Scientific Computing*, 32(1):212–228, February 2010.

[5] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger. Vectorization of an augmented riemann solver for the shallow water equations. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, pages 193–201, 2014.

[6] M. Bader, S. Schraufstetter, C. A. Vigh, and J. Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using sierpinski curves. 4(1):12–21, November 2008.

[7] D.S. Bale, R.J. LeVeque, S. Mitran, and J.A. Rossmanith. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM Journal on Scientific Computing*, 24(3):955–978, 2003.

[8] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2), 2011.

[9] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II — a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, 33(4), 2007.

[10] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2–3):103–119, 2008.

[11] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. *Computing*, 82(2–3):121–138, 2008.

[12] P. Bastian, W. Hackbusch, and G. Wittum. Additive and multiplicative multigrid : a comparison. *Computing*, 60(4):345–364, 1998.

[13] J. Behrens and M. Bader. Efficiency considerations in triangular adaptive mesh refinement. *Philosophical Transactions of the Royal Society A*, 367:4577–4589, October 2009. Theme Issue 'Mesh generation and mesh adaptation for large-scale Earth-system modelling'.

[14] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[15] A. Brandt. Multi-level adaptive technique (mlat) for fast numerical solution to boundary value problems. In *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, number 18 in Lecture Notes in Physics, pages 82–89, 1973.

[16] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31(138):333–390, 1977.

[17] H.-J. Bungartz, W. Eckhardt, T. Weinzierl, and C. Zenger. A precompiler to reduce the memory footprint of multiscale pde solvers in c++. *Future Generation Computer Systems*, 26(1):175–182, January 2010.

[18] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[19] H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. The pde framework peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive cartesian grids. *Computational Mechanics*, 46(1):103–114, June 2010. published online.

[20] H.-J. Bungartz, M. Mehl, and T. Weinzierl. A parallel adaptive Cartesian PDE solver using space–filling curves. In E. W. Nagel, V. W. Walter, and W. Lehner, editors, *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *Lecture Notes in Computer Science*, pages 1064–1074, Berlin Heidelberg, 2006. Springer-Verlag.

[21] C. Burstedde and D. Calhoun. Forestclaw—a parallel, adaptive library for logically cartesian, mapped, multiblock domains, 2015. http://math.boisestate.edu/ calhoun/ForestClaw.

[22] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–15. IEEE Press, 2008.

[23] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[24] B. A. Cipra. The best of the 20th century: Editors name top 10 algorithms. *SIAM News*, 33(4), 2000.

[25] Clawpack Development Team. Amrclaw—part of clawpack-5, 2014. http://www.clawpack.org/amrclaw.html.

[26] S. Cools, B. Reps, and W. Vanroose. An efficient multigrid calculation of the far field map for Helmholtz and Schrödinger equations. *SIAM Journal on Scientific Computing*, 36:B367–B395, 2014.

[27] J. Davison de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: a massively parallel problem solving environment. *The Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41, 2000.

[28] W. Dehnen. A hierarchical o(n) force calculation algorithm. *J. Computational Physics*, 179(1):27–42, 2002.

[29] J. Dongarra, J. Hittinger, et al. Applied Mathematics Research for Exascale Computing, DOE ASCR Exascale Mathematics Working Group, 2014.

[30] A. Dubey et al. The Flash Code, 2015. http://flash.uchicago.edu/site/flashcode.

[31] W. Eckhardt and T. Weinzierl. A blocking strategy on multicore architectures for dynamically adaptive pde solvers. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics, PPAM 2009*, volume 6068 of *Lecture Notes in Computer Science*, pages 567—-575. Springer-Verlag, 2010.

[32] M. Emmett and M.L. Minion. Toward an efficient parallel in time method for partial differential equations. *Communications in Applied Mathematics and Computational Science*, 7(1):105–132, 2012.

[33] Enzo Developers. enzo—astrophysical adaptive mesh refinement, 2015. http://enzo-project.org.

[34] A. C. Frank. *Organisationsprinzipien zur Integration von geometrischer Modellierung, numerischer Simulation und Visualisierung.* Herbert Utz Verlag, Dissertation, Institut für Informatik, Technische Universität München, 2000.

[35] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1st edition, 1994.

[36] M. Gander and S. Vandewalle. Analysis of the parareal time-parallel time-integration method. *SISC*, 29(2):556–578, 2007.

[37] M. Gander and S. Vandewalle. On the Superlinear and Linear Convergence of the Parareal Algorithm. volume 55 of *LNCSE*, pages 291–298. 2007.

[38] P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the gmres algorithm on massively parallel machines. *SISC*, 35(1):C48–C71, 2013.

[39] P. Ghysels, P. Klosiewicz, and W. Vanroose. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra with Applications*, 19(2):253–267, 2012.

[40] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 2014. (in press).

[41] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014.

[42] P. Ghysels and W. Vanroose. Modeling the performance of geometric multigrid on many-core computer architectures. *SISC*, 2015. (submitted).

[43] B. Gmeiner, H. Köstler, M. Stürmer, and U. Rüde. Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. *Concurrency Computat.: Pract. Exper.*, 26(1):217–240, 2014.

[44] P. Gonnet, M. Schaller, et al. Swift—fast hybrid (shared/distributed memory) sph code for astrophysics, 2015. http://icc.dur.ac.uk/swift.

[45] M. Grandin. Data structures and algorithms for high-dimensional structured adaptive mesh refinement. *Advances in Engineering Software*, 82:75–86, 2015.

[46] M. Grandin and S. Holmgren. Parallel data structures and algorithms for high-dimensional structured adaptive mesh refinement. Technical Report 20, Uppsala Universitet, 2014.

[47] M. Griebel. *Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hiearchischen-Transformations-Mehrgitter-Methode*, volume 342/4/90 A. Dissertation, Technische Universität München, 1990.

[48] M. Griebel. Multilevel algorithms considered as iterative methods on semidefinite systems. *SIAM J. Sci. Comput*, 15(3):547–565, 1994.

[49] M. Griebel and G. Zumbusch. Parallel Multigrid in an Adaptive PDE Solver Based on Hashing and Space-filling Curves. *Parallel Comput.*, 25(7):827–843, July 1999.

[50] F. Günther. *Eine cache-optimale Implementierung der Finiten-Elemente-Methode*. Dissertation, published electronically, Institut für Informatik, Technische Universität München, 2004.

[51] F. Günther, M. Mehl, M. Pögl, and C. Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, 2006.

[52] S. Güttel and M. Gander. Paraexp: A parallel integrator for linear initial-value problems. *SISC*, 35(2):C123–C142, 2013.

[53] W. Hackbusch. Parabolic multi-grid methods. In R. Glowinski and J. L. Lions, editors, *Computing Methods in Applied Sciences and Engineering VI*, pages 189–197. North-Holland, 1984.

[54] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[55] J. Hartmann. *Entwicklung eines cache-optimalen Finite-Element-Verfahrens zur Lösung d-dimensionaler Probleme.* PhD thesis, Institut für Informatik, Technische Universität München, 2004.

[56] T. Haut, T. Baab, P.G. Martinssono, and B. Wingate. A high-order scheme for soolving wave propagatoin problems via the direct construction of an approximate time-evoluation operatoor. *IMA Journal on Numerical Analysis*, 2014. submitted.

[57] T. Haut and B. Wingate. An Asymptotic Parallel-in-time Method for Highly Oscillatory PDEs. *SISC*, 2014. (in press).

[58] H. Haverkort, M. Bader, and T. Weinzierl. Space-filling curves for 3d mesh traversals. Talk at ParCo 2013, 2013. http://www.win.tue.nl/ hermanh/stack/h-sfc3dmt-talk.pdf.

[59] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC14*, pages 3–14. IEEE, IEEE, 2014.

[60] W. Herder. Lastverteilung und parallelisierte Erzeugung von Eingabedaten für ein paralleles cache-optimales Finite-Element-Verfahren. Diploma Thesis, Institut für Informatik, Technische Universität München, 2005.

[61] G. Horton and S. Vandewalle. A space-time multigrid method for parabolic partial differential equations. *SISC*, 16(4):848–864, 1995.

[62] J. Janssen and S. Vandewalle. Multigrid waveform relaxation on spatial finite element meshes: The discrete-time case. *SIAM Journal on Numerical Analysis*, 33:456–474, 1993.

[63] M. Käser, C. Pelties, A. Gabriel, et al. Seissol, 2015. http://seissol.geophysik.uni-muenchen.de.

[64] D. E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12. Springer-Verlag, 1990.

[65] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies 2002*, pages 213–232. Springer-Verlag, 2003.

[66] A. Krahnke. *Adaptive Verfahren höherer Ordnung auf cache-optimalen Datenstrukturen für dreidimensionale Probleme.* Dissertation, published electronically, Technische Universität München, 2005.

[67] M. Langlotz. Parallelisierung eines Cache-optimalen 3D Finite-Element-Verfahrens. Diploma Thesis, Fakultät für Informatik, Technische Universität München, 2004.

[68] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[69] J.-L. Lions, Y. Maday, and G. Turinici. A parareal in time discretization of PDE's. *C.R. Acad. Sci. Paris, Serie I*, 332:661–668, 2001.

[70] A. Masud and T.J.R. Hughes. A space-time Galerkin/least-squares finite element formulation of the Navier-Stokes equations for moving domain problems. *Comput. Methods Appl. Mech. Eng.*, 146, 1997.

[71] M. Mehl, T. Weinzierl, and C. Zenger. A cache-oblivious self-adaptive full multigrid method. *Numerical Linear Algebra with Applications*, 13(2-3):275–291, 2006.

[72] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario, 1966.

[73] M. Pögl. *Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens für große Probleme*, volume 745 of *Fortschritt-Berichte VDI, Informatik Kommunikation 10.* VDI Verlag, Dissertation, Technische Universität München, 2004.

[74] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of SC '10*, pages 1–11, 2010.

[75] B. Reps, W. Vanroose, and H. bin Zubair. On the indefinite Helmholtz equation: Complex stretched absorbing boundary layers, iterative analysis, and preconditioning. *Journal of Computational Physics*, 229(22):8384–8405, 2010.

[76] B. Reps and T. Weinzierl. Complex additive geometric multilevel solvers for helmholtz equations on spacetrees. *Journal*, Year.

[77] H. Sagan. *Space-filling curves*. Springer-Verlag, New York, 1994.

[78] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[79] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros. Dendro: Parallel algorithms for multigrid and amr methods on 2:1 balanced octrees. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 18:1–18:12, Piscataway, NJ, USA, 2008. IEEE Press.

[80] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Cluster optimization and parallelization of simulations with dynamically adaptive grids. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 484–496, Berlin Heidelberg, 2013. Springer-Verlag.

[81] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Sfc-based communication metadata encoding for adaptive mesh. In Michael Bader, editor, *Proceedings of the International Conference on Parallel Computing (ParCo)*, October 2013. accepted.

[82] R. Speck, D. Ruprecht, R. Krause, M. Emmett, M. Minion, M. Winkel, and P. Gibbon. A massively space-time parallel n-body solver. Technical report, ICS—Institute of Computational Science, Universita della Svizzera italiana, 2012. Supercomputing '12.

[83] V. Springel et al. Gadget-2—a code for cosmological simulations of structure formation, 2015. http://wwwmpa.mpa-garching.mpg.de/gadget.

[84] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 43:1–43:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[85] H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM J. Sci. Comput.*, 30(5):2675–2708, August 2008.

[86] R. Teyssier. Ramses, 2015. http://www.itp.uzh.ch/ teyssier/ramses/RAMSES.html.

[87] K. Unterweger, T. Weinzierl, D. Ketcheson, and A. Ahmadia. Peanoclaw - a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperbolic conservation law solvers. Technical report, Institut für Informatik, Technische Universität München, June 2013.

[88] W. Vanroose, F. Martin, T. N. Rescigno, and C. W. McCurdy. Complete photo-induced breakup of the H_2 molecule as a probe of molecular electron correlation. *Science*, 310:1787–1789, 2005.

[89] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87:266–290, 1995.

[90] M. Weinzierl. *Hybrid Geometric-Algebraic Matrix-Free Multigrid on Spacetrees*. Dissertation, Fakultät für Informatik, Technische Universität München, München, 2013.

[91] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Verlag Dr. Hut, München, 2009.

[92] T. Weinzierl. The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. Technical Report 2015arXiv150604496W, Durham University, 2015.

[93] T. Weinzierl, M. Bader, K. Unterweger, and R. Wittmann. Block fusion on dynamically adaptive spacetree grids for shallow water waves. *Parallel Processing Letters*, 24(3):1441006, September 2014.

[94] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2015. www.peano-framework.org.

[95] T. Weinzierl and T. Köppl. A geometric space-time multigrid algorithm for the heat equation. *Numer. Math. Theor. Meth. Appl.*, 5(1):110–130, 2012.

[96] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, October 2011.

[97] T. Weinzierl, B. Verleye, P. Henri, and D. Roose. Two particle in tree realisations. *Parallel Programming*, xxx(x):xxx, 2015.

[98] I. Yavneh and M. Weinzierl. Nonsymmetric black box multigrid with coarsening by three. *Numerical Linear Algebra with Applications*, 19(2):246–262, 2012.