TECHNISCHE UNIVERSITÄT MÜNCHEN

Department of Informatics
Scientific Computing

A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids

Tobias Weinzierl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Bernd Brügge, Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Hans-Joachim Bungartz
	2. Univ.-Prof. Dr. Dr. h.c. Christoph Zenger
	3. Univ.-Prof. David E. Keyes, Ph.D., Columbia University, New York/USA

Die Dissertation wurde am 23. März 2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23. Juni 2009 angenommen.

Abstract

In many fields of application in science and engineering, the grid-based numerical simulation of partial differential equations leads to new scientific insights due to increasing computing resources, increasing amounts of data, and increasing efficiency of the algorithms used. All three of them facilitate more detailed models and more reliable simulations. Yet, a growing code complexity accompanies this progress.

To tackle this complexity, more and more solvers for partial differential equations rely on frameworks. State-of-the-art frameworks have to support multiscale algorithms for arbitrary dimensional problems with dynamic, i.e. changing, discretisations. Despite this flexibility of data and data access, the realisation has to have low memory requirements, as the gap between computing power and memory bandwidth and access speed broadens. It furthermore has to exploit modern computer architectures and has to scale on parallel computers, even if the data structures and the data access pattern change permanently.

This thesis presents a framework tackling these challenges with adaptive Cartesian grids resulting from a generalised octree concept. They are traversed with space-filling curves and a small yet fixed number of stacks acting as data containers. Within this context, dynamically changing multiscale grids of arbitrary dimension can be stored with a few bits per datum, whereas classical approaches often require several kilobytes and more flexible data structures entailing a runtime overhead. The thesis formalises the approach and reduces the implementation complexity—the algorithmic principle itself has been well-known for several years—from an exponential to a linear number of containers in the spatial dimension of the problem. A modification of the grid traversal originally following a depth-first order then facilitates a domain decomposition strategy with dynamic load-balancing.

The framework is named after the Italian mathematician Giuseppe Peano who discovered the underlying space-filling curve. Its potential is demonstrated via a geometric multigrid solver for the Poisson equation with low memory requirements, good cache hit rates, a posteriori adaptive grids, and a dynamic load balancing—a combination of characteristics rarely found. As the characteristics result from the framework usage, the thesis paves the way to a multiscale computational fluid dynamics application on instationary, hierarchical grids that uses Peano.

Zusammenfassung

Mit steigender Rechenleistung, steigendem Datenumfang und Datendetailfülle sowie steigender Algorithmeneffizienz liefert die Simulation partieller Differentialgleichungen neue wissenschaftliche Erkenntnisse in vielen Anwendungsbereichen aus Naturwissenschaft und Technik. Dabei sei hier die numerische Simulation auf räumlichen Diskretisierungen thematisiert. Alle vier Einflussfaktoren stoßen die Tür zu immer detaillierteren Modellen und immer verlässlicheren Simulationen auf, jedoch begleitet eine beständig wachsende Quelltextkomplexität eben diesen Fortschritt.

Um genau diese Komplexität in den Griff zu bekommen, greifen mehr und mehr Umsetzungen von Lösern zu partiellen Differentialgleichungen auf Frameworks, also vorgefertigte Quelltextumgebungen respektive -ökosysteme, zurück. Kompetitive, zeitgemäße Frameworks müssen heute Multiskalenalgorithmik für beliebig dimensionale Probleme mit sich ständig dynamisch ändernden räumlichen Diskretisierungen unterstützen. Trotz der geforderten Flexibilität in Daten und Datenzugriff sollte die Realisierung jedoch geringe Speicheranforderungen aufweisen, da sich zwischen vorhandener Rechenleistung und Speicherbandbreite eine Kluft auftut, die sich beständig weitet. Darüber hinaus muss sie moderne Rechnerarchitekturen sinnvoll, also öknomisch und effizient, nutzen und sollte auch dann auf Parallelrechnern skalieren, wenn sich Datenstrukturen und Datenzugriffsmuster permanent ändern.

Diese Arbeit präsentiert eine solche Umgebung, die eben angesprochene Herausforderungen mit adaptiven kartesischen Gittern angeht. Diese entstammen einem verallgemeinerten Oktalbaumkonzept und werden mit raumfüllenden Kurven durchlaufen, wobei eine kleine, jedoch festgeschriebene Zahl an Stapeln als Daten-Container fungiert. In solch einer Umgebung lassen sich sich dynamisch ändernde Gitter mit wenigen Bits pro Datum ablegen. Klassische Ansätze veranschlagen hierzu oftmals mehrere tausend Bytes und verlangen nach alternativen, flexiblen Datenstrukturen, die einen großen Laufzeit-Overhead nach sich ziehen. Die Arbeit formalisiert zum Einen den neuen Ansatz, zum Anderen reduziert sie die Implementierungskomplexität—das algorithmische Prinzip ist seit einigen Jahren wohlbekannt—von einem exponentiellen auf ein lineares Wachstum in der Raumdimension des Problems. Eine Modifikation des Traversierungsprinzips, das originär einer einfachen Tiefensuche nacheifert, erlaubt schlussendlich, eine Rechengebietszerlegungsstrategie mit einem dynamischen Lastausgleich umzusetzen.

Das Framework ist nach dem italienischen Mathematiker Giuseppe Peano benannt, der die zugrundeliegende raumfüllende Kurve entdeckt hat. Der Implemen-

tierung Potential wird anhand eines geometrischen Mehrgitterlösers für die Poisson-Gleichung dargelegt. Dieser weist in Folge dann einen sehr geringen Speicherbedarf in Begleitung sehr guter Cache-Trefferraten auf, wobei auch dynamische Gitterverfeinerung und Lastbalancierung zur Anwendung kommen. Eine Kombination all dieser Charakteristiken ist üblicherweise schwer zu finden. Da sie direkt auf die Benutzung des Frameworks zurückzuführen ist, besteht die berechtigte Hoffnung, dass das selbige den Weg zu einer multiskalen Strömungsdynamikanwendung auf instationären, hierarchischen Gittern ebnet.

Introduction

Numerical simulation of phenomena modelled by partial differential equations is of uttermost importance for new scientific insights and industrial innovation, and the solution of the equations belongs to the grand challenges of many disciplines. Simulation-driven quantum leaps can be found in many fields of application—from Astrophysics illustrating Nature's title page on June 2, 2005, over racing yachts winning the America's Cup in 2003 and 2007, down to Nanophysics predicting black holes created by next generation particle colliders such as the Large Hadron Collider at CERN . Consequently, more and more publications outline the significance of the discipline—the PITAC Report on Computational Science released in 2005 by the the President's Information Technology Advisory Committee [5] is perhaps most prominent—more and more supercomputers relocate the front line of processing capacity, and more and more national and international supercomputing centers and initiatives enter the spotlight of the public attention. Computational science and engineering being the third pillar of research besides modelling and experiment is an omnipresent term.

The foundations of this third pillar is the trinity of data, hardware, and algorithmics. It is kept together by software acting as integrator and catalyst: Software, on the one hand, brings the three disciplines together. On the other hand, it has to exploit synergy effects and make the whole more than the sum of its parts. For the growth of complexity of the individual disciplines and the frictional loss of a naive combination of single techniques and paradigms, this task is far from trivial. Data becomes available in overwhelming quantity—installations such as the Large Hadron Collider need a complete supercomputer network to record, interpret, and post-process the measurements—requiring adaptive multi-resolution, high-dimensional, and distributed data formats. Algorithms exhibit complicated multiscale behaviour incorporating multiphysics models that couple equations describing different phenomena and referring to different spatial scales. Hardware finally becomes massive parallel and tailored for specialised code and single-purpose data streams.

An example illustrates the resulting difficulties: A high performance computer such as SGI's Altix system benefits from simple, homogeneous code processing data sequentially without case-distinctions and jumps in the memory. Due to a data decomposition approach, such a code usually is deployed to the multiple cores running the same execution stream each, it exploits the processors' cache hierarchies due to the sequentiality of the data access, and it fits to the vast amount of registers

without an out-of-order logic due to the lack of case-distinctions. A multi-resolution algorithm such as a multigrid solver though lacks such a homogeneity—the operators change from iteration to iteration, and the algorithm's nature imposes non-local memory accesses. Real-world data such as permeability coefficients of porous media finally make standard multigrid algorithms break down, as the latter rely on an isotropic, homogeneous multiscale behaviour of the underlying partial differential equations. Simulation codes on supercomputers thus often perform with a disappointing speed, lack sophisticated yet well-understood algorithms and algorithmic improvements, or handle dramatically simplified data. In a worst case, a combination of all three aspects.

Keeping all the details from all the disciplines in mind throughout the high performance software development cycle is almost impossible for single developers. At least, it is economically unreasonable because of the time to be spent on sufficiently elaborate implementations if they have to be built from the scratch. Unfortunately, even the software that is already available often lacks a sufficient level of quality, and the PITAC report coins the phrase of a software crisis: *[...] today's computational science ecosystem is unbalanced, with a software base that is inadequate to keep pace with and support evolving hardware and application needs.*

Establishing such a software base and a reuse culture is an enormous challenge covering every issue from the documentation of best practices and design patterns over the programming to standardised interfaces to the black-box reuse of complete code repositories. Two code reuse paradigms coexist and compete: Software is either built bottom-up by a combination of individual toolboxes and black-box components from a library—the user then integrates and composes the separate parts manually or with domain specific high-level languages—or it integrates into frameworks providing a sophisticated and elaborate source code and feature environment. Pros and cons adhere to both approaches with respect to flexibility, extendability, exchangeability of components, and applicability to different problems. Yet, it finally comes down to the question whether the resulting code efficiently implements a sophisticated algorithm on a piece of hardware with the given data sets, while the code features are encapsulated and the implementation obeys the separation of concepts paradigm, i.e. while the code preserves a high quality. In this thesis, I present the C++ framework Peano addressing selected facets of these challenges.

The Spacetree Paradigm in a Framework for Partial Differential Equations

The idea of spacetrees pulls through each algorithmic and architectural design decision of the framework. Spacetrees start from a hypercube; a successive, recursive refinement of this hypercube then yields a spatial discretisation corresponding to an adaptive Cartesian grid. They look back on a long tradition at our group in Munich. Favoured for their simplicity and their multiscale spatial representation, they

have proven of great value for several proof-of-concept implementations of fluid dynamic and fluid-structure interaction codes, geometric multigrid algorithms, solvers dynamically refining the grid where appropriate, and so forth.

It is a common wisdom that spacetrees benefit from space-filling curves: With a spacetree traversal following such a curve, one can encode the spacetree efficiently. Efficiency hereby covers both the amount of memory required and the memory access characteristics with respect to indirect addressing and memory/cache hierarchies. All data structures needed for such a traversal are stacks if the particular space-filling curve is carefully chosen—a fact not obvious to our group before 2003 [35, 63]. Several prototypes confirmed this observation for different spatial dimensions, and several prototypes confirmed that parallelisation, geometric multigrid algorithms, dynamic adaptivity, and so forth also fit perfectly to the spacetree-stack combination—each a piece of software on its own.

This thesis picks up the idea of spacetrees traversed by space-filling curves. It formalises the discretisation principle and generalises as well as simplifies the traversal and grid management algorithm. I end up with an efficient grid management that is able to handle adaptive Cartesian grids, runs in parallel, supports dynamic adaptivity as well as multiscale algorithms, and so forth—I combine, integrate, and extend many features of the individual prototypes available before. Since the traversal acts as an algorithmic blueprint with concrete algorithms plugging into the traversal, this establishes a framework for sophisticated algorithms for challenging partial differential equations.

In the public eye, scientific computing typically comes second behind the disciplines applying high performance computing, since few people are originally interested in methodological, (software-)aesthetic, or even runtime improvements. People are interested in new insights or better understanding. Freely adapted from Richard Hamming, the purpose of computing is neither runtime nor numbers but insight. Alternatively, *[. . .] the ultimate purpose of computing is insight, not petaflop/s [. . .]* [44, p. 2]. A framework's worth consequently becomes apparent as soon as a scientific or engineering code based upon the very framework yields significant results.

I demonstrate the applicability of my framework with a simple matrix-free, multiplicative, geometric multigrid solver for the Poisson equation. Matrix-free at this is the keyword, as matrix-free methods reveal a number of unique selling points that are especially important for many simulations. They are discussed in a moment. Having no global matrices at hand, it is though not trivial anymore to apply direct or iterative solvers or sophisticated preconditioners to improve the solver's convergence. Consequently, the solver of the linear equation system itself has to make up this constraint. Fortunately, multiplicative multigrid algorithms already are optimal solvers for elliptic problems and they play in the upper league for many other problems [78]. While the Poisson solver demonstrates that it is possible to implement such an optimal solver within the framework and that the solver adopts

all the framework's properties, it is far from a real-world problem. For really interesting, real-world problems, I want to refer the reader to subsequent theses and publications—namely [9] and [60].

Selected Challenges in High Performance Computing

This thesis tackles selected challenges high performance computing nowadays faces. Because of the framework approach, the benefits resulting from each solution tackling a challenge carry over for each solver implemented.

From the data point of view, Peano comes along with very low memory requirements. Furthermore, it is not restricted by the available main memory. For dynamic, arbitrary adaptive grids, the memory per vertex ratio in many out-of-the-box solvers exceeds several hundred bytes and restricts the simulation's maximum size of simulation runs. Due to the stack-based containers and the spacetree discretisation, the algorithm here gets along with less than one byte per degree of freedom. This byte holds the complete adjacency, connectivity, and adaptivity information. With such an approach, one can handle problems that are by magnitudes bigger than many conventional simulation runs, and the algorithm is not bandwidth-restricted, i.e. the memory connection does not slow down the code.

Operating systems swap data to the hard disk whenever a code's memory requirements outrun the main memory. Yet, the application's performance then usually breaks down, and the maximum swap data size is usually also restricted by the operating system's installation. In general, simulations have to get along with the available memory. This constraint is particularly dominant for problems suffering under the curse of dimensions or requiring almost regular and very fine grids. Direct numerical simulation of turbulent flows or high-dimensional problems arising in mathematical finances are popular examples. Due to the grid's stack-based persistence management (all data containers are stacks), the algorithm here is able to offer a tailored file swapping strategy temporarily storing data subsets of arbitrary size to disk without runtime penalties. As a result, solely the application's instruction stream and a fixed record buffer have to fit into the memory.

From the algorithmic point of view, Peano's Poisson solver implements a state-of-the-art multiplicative, geometric multigrid with a full approximation storage on the adaptive Cartesian grid, i.e. it holds a solution representation on every grid level. While the implementation follows standard multigrid text books, it highlights four implementation advantages resulting from matrix-free methods—the system matrices are never assembled, instead, the matrix-vector products associated with the multigrid solver are evaluated on-the-fly throughout the grid traversal. First, generating systems and spacetrees fit perfectly together. Switching from a nodal or hierarchical basis to a hierarchical generating system simplifies the multigrid's arithmetics and yields the reference system to compute the coarse grid corrections

for free. Second, it circumnavigates the challenge to elaborate a fitting matrix storage format. The set of linear equations resulting from arbitrary adaptive grids even for simple partial differential equations with simple operator discretisations comprises matrices exhibiting a complicated sparsity pattern, and the realisation or usage of appropriate sparse matrix formats is laborious and error-prone. Third, if the algorithm assembled system matrices, it would need additional memory to hold these data structures. This memory is saved, i.e. solely the grid size determines the algorithm's need for memory. Finally, assembling a system matrix involves a certain runtime overhead disappearing for matrix-free methods. The latter three arguments gain weight if the grid and, hence, the matrices change permanently either due to a dynamic grid refinement or a multiscale algorithm.

From the hardware point of view, Peano exploits modern cluster architectures in three different ways. First, it provides uniform data access cost. To access data stored in the processing unit's registers is by magnitudes cheaper in terms of runtime than fetching data from the main memory (or even hard disk). Modern computer architectures mind this fact and introduce a cascade of caches holding small sets of copies from the main memory. Accessing these copies is significantly faster than the access to the main memory, i.e. the runtime penalty for accessing non-register values is reduced if the algorithm does not read from or write to the main memory directly. As a result, the runtime per record access is not constant, but it depends on the record's location. Due to the space-filling curves and the stack-based data containers, Peano's traversal restricts to cache and register accesses. The number of non-cache accesses, i.e. cache misses, is negligible, and the runtime per record is constant. Peano is cache-oblivious. This algorithmic ingredient goes hand in hand with the fact that the hard disk swapping does not thwart the simulation code: the cost per record remains constant, even if the memory requirements exceed the main memory.

Second, Peano provides a domain decomposition strategy deploying the spacetree among multiple computing nodes. The resulting advantage is twofold: On the one hand, the individual nodes run in parallel, i.e. the performance improves. On the other hand, the individual nodes require less memory, i.e. if the experiment would exceed the (hard disk and/or main) memory of one computing node, several nodes in combination cope with the problem. Throughout the decomposition, the space-filling curve ensures that the subpartitions are quasi-optimal, i.e. the data that is to be exchanged to couple the smaller problems on the individual nodes is minimal relative to the workload, and this data exchange is straightforward to implement, i.e. it does not induce an additional sorting or mapping step. As network interconnections are bottlenecks of a parallel computers, modest exchange package sizes are essential for parallel codes. As the exchange process integrates smoothly into the grid management without any sophisticated reordering or synchronisation, the homogeneous data access cost moreover are preserved.

Third, Peano provides load balancing. It distributes the workload equally among the computing nodes, i.e. it tries in a greedy fashion to assign each node the same amount of computations. The balancing's realisation is itself parallelised, i.e. the balancing itself scales with the parallel nodes and does not become a bottleneck. Such a property is essential for massive parallel architectures. Furthermore, the balancing works on-the-fly, i.e. it permanently adopts and optimises the partitioning. This is essential for dynamic adaptive algorithms.

The splitting into three different classes of features coincides with many publications, and the tackled challenges can be found in a vast amount of literature. Peano queues into this group. Low memory requirements draw through the discussion of advantages of many structured grid approaches including adaptive mesh refinement methods with regular patches. Matrix-free methods, cache-oblivious algorithms, parallelisation and load balancing are popular subject of attention of many multigrid solver implementations. Two important aspects of realisations in high performance computing though are not covered at all: On the one hand, the integration of legacy or third-party code into the Peano framework is beyond the scope of the thesis. Although it is inconsistent to motivate a framework development with code reuse and separation of concerns without reusing software, I skip this discussion and develop most of the realisation's ingredients from skratch. For tests, validation, and quick prototyping, connecting to software not tailored to the spacetree world is surely reasonable and routine [60]. To exploit Peano's full properties and to preserve Peano's advantages, most codes presumably have to be adopted. I assume that white box reusage copying code blocks, algorithmic ideas, and best practices outshines black-box code reuse. Consequently, there is a need for reusage and tailoring patterns and strategies, and there is also a need for a careful elaboration where third-party component usage nevertheless does make sense. This reasoning is picked up in the conclusion. On the other hand, many papers in scientific computing concentrate on the development of more and more elaborate and sophisticated algorithms. They derive schemes yielding qualitatively better results per atomic computing operation or per record beyond the result improvements due to an increased computational effort, i.e. beyond just scaling up the problem size. The term *more science per flop* [44, p. 14] perhaps describes this aspect best. Obviously, such an endeavour is beyond the foundations of the pillar of scientific computing, and, hence, beyond the scope of this work.

While each of the tackled challenges is an interesting subject of study itself, the combination of all of them is the outstanding highlight of the framework. Because of the strict separation of the individual packages and concerns, any solver programmed along the ideas of the Poisson solver carries over the same properties. This is a promising insight making me believe that the framework can be the basis of simulations tackling challenges not solved before.

Alternative Frameworks

This vision underlies many frameworks and libraries. While a well-grounded comparison and evaluation of different code packages is beyond the scope and volume of this thesis, I nevertheless pick out three alternative projects here and classify Peano with respect to them. Besides a better impression what Peano's characteristics and unique selling points look like, such a classification also gives hints in the conclusion what features Peano is missing and what Peano has to learn from other endeavours. For the time being, I concentrate on highlights, similarities, and common approaches. The three non-commercial reference systems are chosen subjectively, i.e. their selection is not exhaustive.

The Distributed and Unified Numerics Environment toolbox DUNE is a first subject of study [3, 4]. DUNE follows a strict separation-of-concept approach due the definition of C++ interfaces, i.e. it defines a toolbox as set of interfaces and delivers several implementations of these interfaces. Programming versus interfaces allows the developer to exchange both individual (sub)algorithms and selected data structures without modifying the complete code basis. Especially interesting is the idea to replace subgrids by optimised, structured patches and to hide the distribution of the grid parts. Both features are hidden from the user, as the algorithms' realisations rely solely on an iterator concept.

Peano and its extensions provide a similar concept—regular patches are optimised and the parallel realisation details are encapsulated from the algorithms—while it relies on a callback mechanism, i.e. the algorithm does not actively traverse the grid data structures, but the grid calls back the algorithms. My implementation lacks the flexibility of unstructured grids. However, it provides arbitrary, dynamic adaptivity—a feature that usually is difficult to provide with regular patches—it provides a multiscale representation of the computational domain, and it provides a fine granular load balancing, where the individual elements and not whole patches are atomic work units. It would be interesting to quantify DUNE's overhead due to the flexibility, to evaluate the benefits arising from unstructered grids, and to elaborate whether using optimised regular patches as black-box can overtake Peano's holistic, structured, and inherent hierarchical approach. Furthermore, DUNE's flexibility facilitates legacy code reuse. While code reuse is laborious for Peano if the old code does not fit to the underlying spacetree paradigma, it is nevertheless desireable in many places, and reuse best-practices hereby are of great value. I pick up this aspect in the conclusion.

Next, I want to mention the Adaptive Large-scale Parallel Simulations (ALPS) library underlying [18, 19]. These papers employ adaptive mesh refinement methods to study mantle convection which is a multiscale problem in both time and space: they solve a real-world problem on a petascale supercomputer. The underlying computations scale on massive parallel environments due to the usage of octrees

and space-filling curves. The octrees here act as key structure for data access, and refining, coarsening, and load decomposition is directly interwoven with this data structure.

All of Peano's features rely on k-spacetrees—a generalisation of the octree concept—and, thus, many paradigms and algorithmic approaches exhibit similarities (the initial setup of the load decomposition, e.g., follows a tree-based top-down approach in both codes). I am sure that these similarities rest upon fundamental principles and patterns of any "spacetree code", and, hence, Peano can learn from ALPS. Nevertheless, exploiting the spacetree directly for the matrix-free PDE solver is a fundamental idea of the Peano framework, while ALPS concentrates on the grid management and its applications employ algebraic solvers. For applications where geometric multigrid solvers can be applied, it would be interesting to compare Peano's holistic approach with applications built on top of ALPS, to quantify the overhead arising from the algebraic approach, and to study the runtime implications of both approaches.

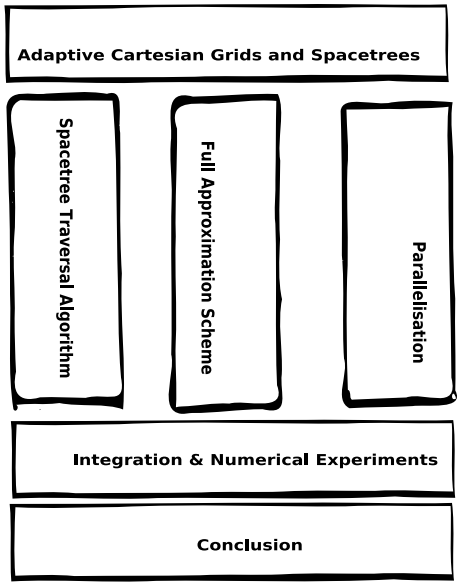
Finally, the PDE solver Hierarchical Hybrid Grids [26, 29] shall be mentioned. It is a matrix-free multigrid solver, i.e. the operations are embedded into the grid traversal, for elliptic problems. It starts with a coarse, conform, unstructured mesh, distributes and balances this mesh, and then refines the individual mesh elements. Outstanding is the enormous number of unknowns the solver is able to handle in combination with its high performance and good scalability. From my point of view, this is in particular due to a unique and intelligent combination of multigrid algorithms, efficient programming techniques, and realisation patterns rarely found in one place.

Peano's extensions also support patches of regular refined grids, and its regular refined subdomains resemble HHG's structured patches. Consequently, I am sure that efficiency patterns and best practices from the hybrid code also can be applied to Peano's implementation. Such a tuning makes the code exploit current computer architectures better. In turn, it allows to compare and quantify the performance drawbacks resulting from Peano's flexibility due to the arbitrary dynamic refinement with a highly optimised PDE solver implementation.

Thesis Structure

The thesis' structure roughly follows the challenge enumeration. After a preamble, I start with a description and formalisation of the spacetree idea in Chapter 2. Besides the interplay of spacetrees, adaptive Cartesian grids, and the continuous computational domain, the first chapter establishes a common language for the subsequent text, and it establishes the concept of the traversal events. They act as plug-in points for algorithms built on top of the framework. Three chapters covering the traversal algorithm (Chapter 3), the multigrid solver (Chapter 4), and the parallelisation (Chapter 5) follow. Wherever possible, they do not depend on

each other, as the separation-of-concerns idea prohibits a strong coupling of the different subjects. Besides some properties resulting from the interplay of individual chapters, these chapters can basically be read and understood in arbitrary order. After some experiments integrating the individual ideas in Chapter 6, one short chapter closes the thesis, draws some conclusions, and presents expectations and preliminary results from further, future, and extending work. The text is framed by the Chapter 1 "Peano in a Nutshell" acting as preamble as well as an appendix. The preamble outlines the thesis' content from a programmer's point of view—a programmer that has to implement a multigrid solver within the framework.



The four big Chapters 3, 4, and 5 exhibit a similar internal construction. An introduction summarises the aim, rationale, and reasoning of the chapter's content. It outlines the basic ideas, compares them to alternative and older results, and points out which subjects are not discussed deliberately. Finally, it describes the chapter's structure. Several sections built upon each other follow. Most outline their content and insights at the beginning. It thus is possible to skip individual parts of the thesis. Some experiments study the framework's properties chapter by chapter. The splitting of the experiments simplifies the identification of cause-and-effect chains. A short outlook finally gives extensions and links.

Acknowledgments

This work has been conducted over a period of four years at the Chair of Scientific Computing in Computer Science of the Technische Universität München. Throughout this time, several research activities such as conferences, graduate workshops,

and research visits abroad were co-funded by the university's International Graduate School of Science and Engineering (IGSSE).

Special thanks are due to Prof. Dr. Hans-Joachim Bungartz and Prof. Dr. Dr. h.c. Christoph Zenger. The idea to traverse a spacetree with the Peano space-filling curve originates from Christoph Zenger, and he and his research group first elaborated ideas, prototypes, and concepts underlying many of my thoughts written down here. His personality, enthusiasm, and encouragement made me decide to do a doctorate. My supervisor Hans-Joachim Bungartz then provided the inspiring environment and ecosystem in his research group to evolve the algorithmic ideas further, to integrate hitherto separated concepts into one code, and to continue to elaborate additional ideas and visions. His mentoring and advice significantly shaped the thesis at hand. Last but not least, his support and personality made me decide to continue research beyond a doctor's degree.

Albeit this text comprises my personal ideas, many persons were involved in the development process, the elaboration of alternatives, the formulation and publications of the results, and the day-to-day business accompanying research at a university. Among them, I particularly want to thank Dr. Alexander Kallischko and Dr. Miriam Mehl. Both volunteered to proof-read the whole manuscript, and the text for sure profits to a great extent from their corrections, their critical analysis, and their constructive annotations. Miriam Mehl also headed the computational fluid dynamics group both giving my ideas and software developments a home and an application and in turn inspiring the code's evolution. It is difficult to appreciate the significance of such a productive environment accordingly. She played a big part in this environment. Among the research group, I especially want to thank Tobias Neckel. Prior to all other codes, his research used my algorithmic ideas and framework for a real-world problem. He thus became the person always first suffering from all the confusions, permanent modifications, and meanders coming along with the development of a big piece of software. His insights and feedback particularly coined the shape of this work, and the fruitful cooperation with him was essential for the success of the group's research. I finally want to thank all the students and colleagues having joined the research group later, as well as all the members of the Chair of Scientific Computing in Computer Science in Munich.

Contents

List of Relevant Publications

M. Langlotz, M. Mehl, T. Weinzierl, and C. Zenger. SkvG: Cache-Optimal Parallel Solution of PDEs on High Performance Computers Using Space-Trees and Space-Filling Curves. In A. Bode and F. Durst, editors. *High Performance Computing in Science and Engineering, Garching 2004*, Springer-Verlag. pp. 71–82, 2005

M. Mehl, T. Weinzierl, and C. Zenger. A cache-oblivious self-adaptive full multigrid method. In R. D. Falgout, editor. *Numerical Linear Algebra with Applications*, volume 13(2-3), pp. 275–291, 2006

H.-J. Bungartz, M. Mehl, and T. Weinzierl. A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves. In W. E. Nagel, W. V. Walter, and W. Lehner, editors. *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, Lecture Notes in Science and Engineering, volume 4128, pp. 1064–1074, 2006

M. Brenk, H.-J. Bungartz, M. Mehl, I. L. Muntean, T. Neckel, and T. Weinzierl. Numerical Simulation of Particle Transport in a Drift Ratchet. In C. Johnson, D. E. Keyes, and U. Rüde, editors. *SIAM Journal of Scientific Computing*, volume 30(6), pp. 2777–2798, 2008

M. Mehl, M. Brenk, I. L. Muntean, T. Neckel, and T. Weinzierl. A Modular and Efficient Simulation Environment for Fluid-Structure Interactions with Large Domain Deformation. In M. Papadrakakis and B. H. V. Topping, editors. *Proceedings of the Sixth International Conference on Engineering Computational Technology*, Civil-Comp Press, Kippen, Stirlingshire, United Kingdom, 2008

I. L. Muntean, M. Mehl, T. Neckel, and T. Weinzierl. Concepts for Efficient Flow Solvers Based on Adaptive Cartesian Grids. In S. Wagner, M. Steinmetz, A. Bode, and M. Brehm, editors. *High Performance Computing in Science and Engineering*, Springer-Verlag, pp. 535–550, 2008

M. Mehl, T. Neckel, and T. Weinzierl. Concepts for the Efficient Implementation of Domain Decomposition Approaches for Fluid-Structure Interactions. In U. Langer, M. Discacciati, D. E. Keyes, O. B. Widlund, and W. Zulehner,

editors. *Domain Decomposition Methods in Science and Engineering XVII*, Lecture Notes in Science and Engineering, volume 60, pp. 591–598, 2008

1 Peano in a Nutshell

In a Nutshell is a preamble. Outlining the content of the individual chapters, stating the important results, and highlighting the underlying vision and rationale, it runs briefly through the whole thesis. Since the thesis establishes a framework for grid-based solvers for partial differential equations (PDEs), its descriptions and explanations are often abstract, technical, and formal. I even outsourced concrete applications and demonstration challenges into the conclusion or refer to other theses and publications. This chapter in turn refrains from all the technical details. I take up the position of a developer implementing a multigrid finite element solver with d-linear shape functions for the Poisson equation, and I show how such a solver can benefit from the framework Peano.

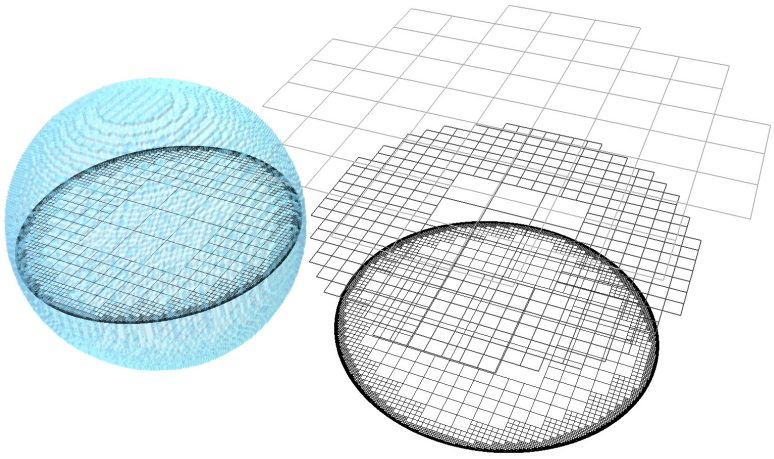


Figure 1.1: Peano solves a PDE for a given compuational domain on an adaptive Cartesian grid (left, cut through a spherical domain). Yet, it does not hold the Cartesian grid in the memory, but it stores the spacetree instead (right).

Before the finite element method breaks down the PDE into a linear equation system, one has to define a discretised representation of the continuous domain—a grid. Throughout the coding, the grid's shape, its representation in the computer, and the available operations on the grid are important. The shape uniquely defined

by the grid construction process determines the numerical properties of the solution. The data structure storing the grid (co-)determines the memory requirements of the implementation, as well as the efficiency of grid and data reads and writes. The operations on the grid—the signature of the grid data structure—split up into two groups: On the one hand, the Poisson solver reads, interprets, and writes back grid-assigned data such as the solution's approximation. On the other hand, the solver modifies the grid itself, i.e. it refines and coarses it. The signature thus influences the execution speed, too, as it freezes the set of operations an algorithm can invoke. Furthermore, this set determines the set of tools available for the implementation.

Before I jump into a grid discussion, the concept of k-spacetrees is introduced (Section 2.1): A single hypercube encloses the computational domain. This hypercube then is recursively refined until the resulting cubes approximate the computational domain with sufficient accuracy. Such a refinement process yields a tree data structure standing for a cascade of smaller and smaller hypercubes. The cascade in turn mirrors an adaptive Cartesian grid, i.e. k-spacetrees give special types of adaptive Cartesian grids. k-spacetrees moreover define several adaptive Cartesian grids with different spatial resolutions simultaneously (Figure 1.1). The principle is simple: Each leaf's hypercube defines a geometric element of the Cartesian grid. All the leaves together tessellate the computational domain. As refined nodes also represent a hypercube, different resolutions of the tessellation, i.e. different Cartesian grids, are given by different levels of the spacetree. Whereas the finest adaptive Cartesian grid is used to apply the finite element method, the coarser grids are useful to implement a multiscale solver.

As soon as a grid is at hand, the solver has to run through it to exploit its information and operate on the data—it traverses the grid. I reveal a tree traversal being a special case of an element-wise grid traversal (Section 2.4 and Section 2.5), i.e. as soon as a tree traversal is found, I can implement an element-wise algorithm on adaptive Cartesian grids. The depth-first traversal is of special interest, as it allows to encode the grid data structure with one bit per element: if the traversal runs top-down through the tree, it has to read one bit per hypercube. This bit determines whether the cube is split up further. I end up with an element-wise algorithm on an adaptive Cartesian grid that comes along with little memory spent on the grid structure. Further, it is simple to enlarge or shorten k-spacetrees. Adding additional elements to the tree equals a grid refinement, removing equals a coarsening. Consequently, I end up with a low-memory algorithm that also supports dynamic adaptivity.

The PDE is defined on a computational domain. A mapping from the continuous domain to the grid is next presented (Section 2.7). It makes all spacetree elements lying inside the domain represent the discretised domain—the continuous domain shrinks to its discrete representation. While such a mapping approximates an arbitrary domain just with first order, it is straightforward to reduce any discreti-

sation error by refining the spacetree further: the approach accepts the low order boundary approximation, as a refinement and, hence, improvement of the boundary approximation is that cheap and simple.

Knowledge about the grid structure is important for the Poisson solver, the visualisation, the data postprocessing, as well as other algorithms. Yet, the actual storage of the tree and the realisation of the traversal is not of interest for the Poisson solver as long as all information is accessible on demand. While one could introduce an access signature on the data structure, I prefer a callback mechanism (Section 2.8): The grid traversal runs through the data structure. For each traversal transition, it triggers an operation—an event—parametrised by the transition's state, the current spacetree element's data, and its vertices. The algorithm plugs into these events, reads, evaluates, and modifies the arguments. A plug-in implementing an error estimator, e.g., listens to the event triggered whenever a vertex is read for the very last time throughout the traversal. At one point, it tells the vertex to refine all its adjacent spacetree elements. In the subsequent iteration, the plug-in then receives the events for the newly created vertices. The refinement realisation, the data structure extension, and the mapping from the continuous domain to the modified grid though are hidden from the plugin. Besides the information hiding, such a mechanism offers additional advantages: a modification or exchange of the solver does not affect the traversal or data structure realisation. At most, the mapping from events to callback routines is altered, and, hence, the user can comfortably combine several algorithms. In the experiments, e.g., I combine an error estimator, a PDE solver, and a visualisation plug-in throughout one single grid traversal.

From now on, chapters have to tackle two orthogonal questions: On the one hand, they have to demonstrate that the plugin mechanism is sufficient to implement a sophisticated solver, i.e. the information hiding and encapsulation do not prohibit the realisation. On the other hand, they have to discuss the actual storage format of both the grid and data assigned to it in context with the efficient structure encoding. I start with the second issue.

A grid consists of vertices, hyperplanes connecting these vertices, and geometric elements, i.e. volumes, in-between. k-spacetrees yield the geometric elements, but the solver for the Poisson evaluates data assigned to the vertices. While the preceding chapter specifies what the traversal looks like—it mirrors a depth-first order—and how the grid's structure is encoded, it neither gives the containers holding the encoding, nor does it refer to the vertex data. I concentrate on these questions and derive a data container realisation going hand in hand with a sophisticated depth-first traversal (Chapter 3). The realisation preserves the low-memory characteristics, it does not restrict the dynamic adaptivity in any way, and it is efficient. The latter aspect covers the algorithmic complexity (the data access is in linear time) and architectural principles, i.e. the realisation fits to and benefits from standard computer architectures. Previous to that an excursus: The Peano space-filling curve makes

the depth-first traversal on a k-spacetree based upon three-partitioning deterministic, as it orders the children of each refined hypercube (Section 3.1). Step by step, the realisation benefits from the nice mathematical properties of Giuseppe Peano's curve[1].

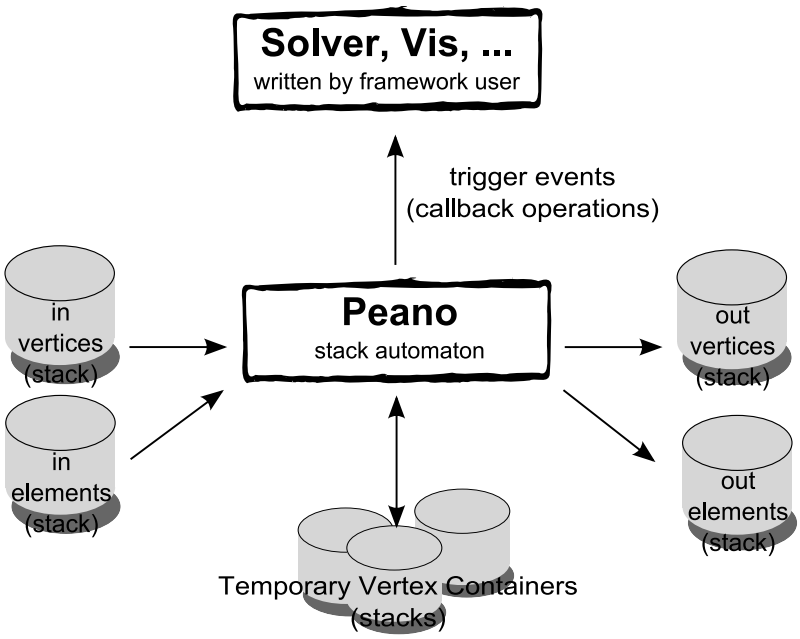


Figure 1.2: Peano's grid persistence management relies exclusively on stacks as data containers. Throughout the grid traversal, it invokes *events*. The solver, visualisation software, data preprocessing component, and so forth plug into these events.

This process is tripartite (Section 3.3): First, I study the handling of the geometric elements. If the depth-first traversal inverts the space-filling curve's order after each iteration, the inverted order again fits to Peano's curve description, and two stacks are sufficient to store all the geometric elements of the k-spacetree—the realisation of an element container is straightforward. Second, I study the order of the vertices before and after a tree traversal. Let a vertex a be used the last time before a vertex b throughout an iteration. Due to the space-filling curve, b will be used for the first time before a throughout the subsequent iteration, and two stacks are sufficient to store all the vertices of the k-spacetree—the realisation of a vertex container is straightforward, too. With these input and output containers, vertices finally have to be managed throughout the iteration: They are read from the input stream the first time they are used. They are stored to the output stream the last time they are

[1] and, finally, the framework adopts the inventor's name

used. They have to be hold in a container in-between these two actions. Again, the space-filling curve allows the realisation to rely on a fixed linear number of stacks for these temporary containers. The number is independent of the grid hierarchy, and, in the end, a finite number of stacks is sufficient to hold all the data throughout the traversals (Section 3.3 and Figure 1.2).

Space-filling curves and stacks are nice subjects of scientific study. With respect to the solver, they have two especially interesting properties: Their implementation comes along with almost no memory overhead, and their data access fits to modern computer architectures: Each stack provides read and write access to only one single position in memory, and this position shifts at most by one entry per data access. Since the number of stacks is fixed, this yields an impressive cache hit rate, and the runtime per vertex becomes independent of the problem size (Section 3.4).

Some realisation details—due to the strict stack-based realisation, one can even handle problems that do not fit into the main memory anymore, and if algorithms evaluate only information up to a given tree level, temporarily removing tree levels below speeds up the traversal substantially—close this discussion. All the data structures, signatures, and implementation concepts for the solver of the Poisson equation are now available.

I hence switch from the realisation to the actual solver (Chapter 4). Due to the inherent multi-resolution property of the k-spacetree, it is obvious to focus on a multigrid algorithm here. After some basic remarks on the finite element method, three topics are discussed. The text argues how k-spacetrees represent a finite function space, what an efficient solver for the PDE's linear equation system does look like, and how the traversal events are mapped to solver operations.

My finite element implementation relies on a d-linear approximation of the continuous function space. Each spacetree vertex yields one basis function of this approximation, and its support covers the 2^d adjacent geometric elements. This approach ends up with a generating system instead of a basis, i.e. the representation of a function is not unique (Section 4.1). To avoid confusion due to ambiguousness I standardise the representations by convention, and, hence, discuss two variants: A nodal scheme holding a function in different spatial representations simultaneously, and a hierarchical scheme splitting up the function into contributions of different frequency.

With the nodal scheme, one can evaluate the method's matrix-vector products element-wisely on the grid: there is no need to set up any global system matrix explicitly, and the resulting approach thus needs no memory in addition to the pure grid data. Next, the matrices are discussed (Section 4.2), and, starting from a simple Jacobi scheme, I then run through the idea of multiplicative geometric multigrid solvers and full approximation schemes with k-spacetrees fitting to the geometric approach, since they hold grids of different spatial resolution by construction. The solution representation with different frequencies simplifies the restriction of the

right-hand side within the multigrid cycle. I thus switch regularly from a nodal representation to the hierarchical scheme.

The individual steps of the multigrid solver plug into the traversal events: the grid and traversal management are encapsulated from the multigrid solver and vice versa. I consequently formulate the solver steps in terms of operations triggered by the single events (Section 4.4). As the algorithm does not introduce additional data or data accesses, it ends up with all the nice properties of the pure traversal—low memory requirements, good cache hit rates, arbitrary adaptivity, and so forth—while it implements a state-of-the-art multiscale solver. Some realisation details and experiments document this.

High performance computing nowadays is dominated by massive parallel computers. The thesis eventually introduces a parallelisation concept based upon a domain decomposition (Chapter 5). It tackles four challenges: An (extended and modified) traversal has to be able to handle a spacetree split into several parts. The realisation of this traversal has to preserve the original traversal's nice properties. And, a load balancing fitting to the traversal has to find a well-suited partitioning, and it has to adopt it the partitioning a changing refinement structure if the Peano solver employs an a posteriori refinement criterion.

First, I show that a depth-first traversal is inherently sequential, i.e. it does not profit from a decomposition of the spacetree. Consequently, I weaken the depth-first paradigm (Section 5.1) and make it a level-wise depth-first traversal preserving all the nice properties, as the stack concept remains exactly the same, and the order of the traversal events is not modified, too.

Second, I discuss the traversal's realisation and the vertex exchange. Both the domain decomposition and the exchange pattern benefit from space-filling curves (Section 5.2): As for the stack-based data management, the curve makes any reordering of exchanged data unnecessary. Furthermore, partitions following the iterate exhibit a small surface compared to their volume, i.e. a nice computation-communication ratio: The partitions are quasi-optimal due to the Hölder continuity of the Peano curve.

The parallel tree traversal postulates properties of a good partitioning. It becomes obvious which partition layout leads to a well balanced decomposition. I third use the properties to derive an on-the-fly load balancing for k-spacetrees (Section 5.3). All traversals permanently keep track of the computational load assigned to one computing node. Furthermore, they observe how many additional work units could be handled by the computing node without becoming a bottleneck for the overall application. Whenever a computing node would become a bottleneck due to a refinement, it takes additional nodes from a global node pool and deploys work to them.

Some remarks on the node pool continue the discussion (Section 5.4). Finally, I combine the solver's actions and the data exchange—due to the exchange of data

the mapping from events to solver operations is slightly to be modified—and end up with a parallel multigrid solver with dynamic load balancing on permanently changing grids.

2 Adaptive Cartesian Grids and Spacetrees

Peano is a framework for grid-based methods for partial differential equations (PDEs), i.e. it relies on a spatial discretisation of the computational domain—a mesh or grid. This chapter defines the grids underlying the whole thesis: it defines the grid structure and the grid generation process, as well as the interplay of the grid and the differential equation's continuous domain.

The spatial discretisation is based on k-spacetrees. They are a d-dimensional generalisation of the octree concept. While octrees rely on bi-partitioning, k-spacetrees employ a k-partitioning: The computational domain is embedded into a hypercube. Then, the discretisation algorithm recursively cuts the cube into k^d equally sized subcubes. While such a discretisation is equivalent to a subset of the class of adaptive Cartesian grids, the efficiency of each algorithm in this thesis exploits the particular spacetree's structure.

Throughout the spacetree discussion, several alternative representations and interpretations are enlisted, but the list is neither complete nor representative. The aim of this chapter instead is as follows: It establishes a uniform terminology and a formalism, to make the text understandable and consistent without additional literature. And, the chapter formally derives algorithmic restrictions and insights for the k-spacetree.

My dissertation looks back to a long tradition of publications discussing challenges, algorithmic ideas, and formalisms coming along with spacetrees. Nevertheless, the underlying ideas are far from being exhausted. Compared to the direct predecessors [35], [63], and [39][1], this chapter picks up facts scattered among the theses, and it harmonises the description. In return, I omit technical details, if they are not essential to understand the algorithmic ideas. Starting from a well-defined terminology, the chapter furthermore states several properties of k-spacetrees or algorithms on k-spacetrees, not written down explicitly before.

The chapter is organised as follows: I first introduce the concept of the k-spacetree in the context of adaptive Cartesian grids with regularity constraints. They are two alternative interpretations of one idea deducing a spatial discretisation for a given domain. In Section 2.3, a formalism describing the grid constituents and their rela-

[1] This is the chronological order.

tionships is given. Such a formalism is essential for algorithms. All algorithms on Cartesian grids require a well-suited traversal of the data structures. Section 2.4 introduces the fundamental ingredients of Peano's element-wise traversals. They then are transferred to the spacetree world in Section 2.5. The subsequent section analyses the traversal's implications and constraints for for algorithms, and it establishes a refinement concept for the adaptive Cartesian grids, i.e. it sets up the data flow underlying the grid generation. Section 2.7 picks up the PDE challenge again, and it discusses the mapping of the continuous domain to the spatial data structures. Next, I formalise the interplay of the k-spacetree , the k-spacetree's traversal and any algorithm making use of these traversals. Such a formalism is important for work extending this thesis ([23, 51, 60], e.g.). A small number of experiments finally highlights some of the k-spacetrees's properties, and an outlook closes the chapter.

2.1 k-spacetree

Algorithms are by definition executable by a machine. Ergo, they need a finite representation of the computational domain Ω. The plenitude of existing geometry representation schemes makes it impossible to give a comprehensive overview: Analytical descriptions compete with surface descriptions such as $\mathcal{B}$-splines or triangulated meshes. Volume-based methods (voxel discretisations or space partitioning data structures, e.g.) face constructive solid geometry models. Surface reconstruction algorithms on data obtained from measurements also might act as geometry representation. A starting point for an overview is for example [68, 72]. Since numerical methods approximate the solution of a partial differential equation—they do not derive an exact, point-wise solution—it is sufficient to approximate the continuous domain. Yet, the geometric error must not pollute the approximation.

Grid-based methods are based on a spatial discretisation of the computational domain, i.e. Ω's representation in the computer is cut into a finite set of primitives—a *grid* or *mesh*. The term *tessellation* is a synonym not highlighting that the domain's boundaries typically are approximated, too. The term grid in turn usually circumscribes the tessellation of the computational domain together with the geometric primitives arising from this discretisation, i.e. the vertices, the primitives, and so forth. I use the terms grid, tessellation, and spatial discretisation as synonyms. The result is Ω_{h}. Formally, the transition $\Omega \mapsto \Omega_{\mathrm{h}}$ is a two step cascade: It first approximates Ω and then cuts it into pieces. Here, both steps are merged in a *spatial discretisation* stemming from a *regular recursive k-section*[2]. Four principles circumscribe k-section:

[2] *Recursive decomposition* is another synonym [67].

—Terminology—

For the k-spacetree, the following terms are well-defined:

The computational domain is embedded into a hypercube. This hypercube equals the *root* of the k-spacetree.

If a cube e_1 is a child of a cube e_2, e_2 is the *parent* of e_1. I write $e_1 \sqsubseteq_{\text{child}} e_2$.

If two cubes e_1 and e_2 share a common parent, e_1 and e_2 are *siblings*.

If two cubes e_1 and e_2 share at least one common point $x \in \mathbb{R}^d$, they are *neighbours*.

If the recursive k-section generates a cube e_1 out of e_2 with an arbitrary positive number of recursion steps, e_1 is a *descendant* of e_2. All children of e_2 are descendants. Not all descendants are children.

The *height* of a k-spacetree equals the recursion depth.

1. The algorithm embeds the computational domain into a hypercube[3]. Since a domain is a bounded, open subset in $\mathbb{R}^d$, this is always possible.

2. For each coordinate axis $x_i, i \in \{1, \ldots, d\}$, there is a hyperplane with a normal along x_i. The algorithm cuts the hypercube into k equally sized parts along each suitably translated hyperplane.

3. The algorithm ends up with k^d small hypercubes. They are disjoint, the sum of their volumes equals the original cube's volume, and all k^d cubes have the same size.

4. The algorithm continues recursively for each new small hypercube if appropriate.

The resulting spatial discretisation is a *k-spacetree*: Each hypercube of the discretisation process corresponds to a vertex in the tree graph. The hypercube into which the computational domain is embedded into is the tree's root. When a cube is cut into k^d pieces, each new subcube is a *child* of the original cube, i.e. there is a directed edge from the bigger to the smaller hypercube.

[3]Hypercubes in this thesis are closed.

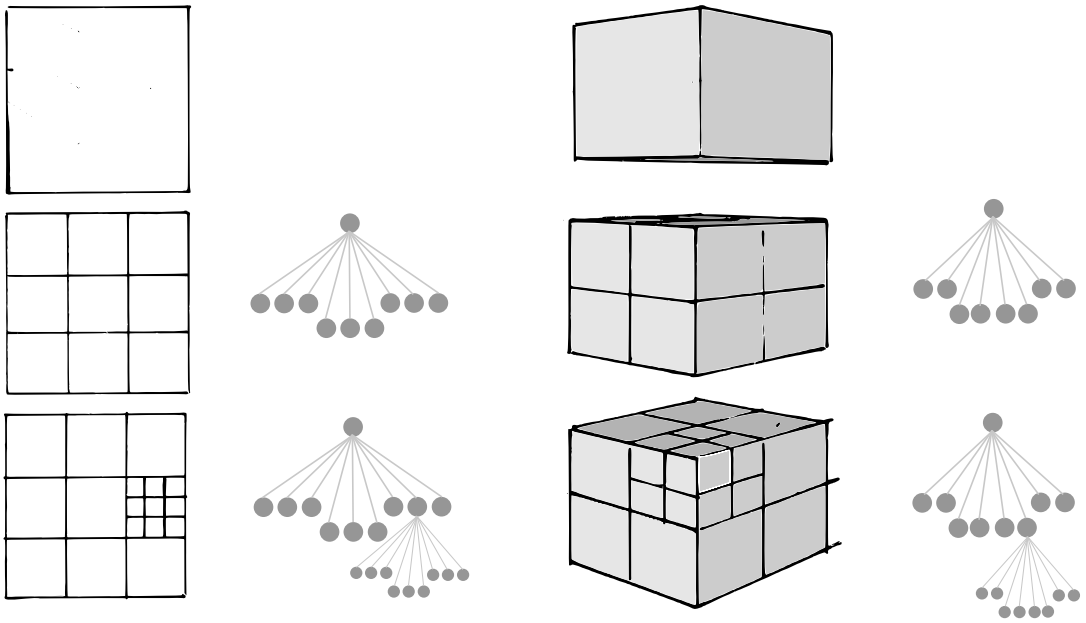


Figure 2.1: k-spacetree construction for $d = 2$ and $k = 3$ (left), and k-spacetree construction for $d = 3$ and $k = 2$ (right) with accompanying tree graph. Both trees at the bottom have height two.

Different recursive k-section variants are well-known under different names in different disciplines. The term *quadtree* for $d = 2$ and $k = 2$ and the term *octree* for $d = 3$ and $k = 2$ (Figure 2.1) are popular names in computer graphics and grid generation (see for example [25, 58, 67, 72]). *Refinement tree* [56], *Q-tree* and *quadtrie* [67] as well as *region quadtree* [68] are among the alternative names. If the cuts are not equidistant or the number of cuts per dimension per cube does not equal an constant k, data structures such as *binary space partitioning trees*, $k-d$ *trees* or *point quadtrees* [20, 67, 72] arise. Some authors generalise the concept and replace the initial cube by another primitive such as a triangle ([2, 56], e.g.). Others introduce different names such as *edge quadtree*, *MX quadtree* or *PM quadtree* for different recursion termination criterion [68]. This list is neither complete nor are the examples representative.

I draw my attention to a fixed number k of equidistant cuts per dimension. The dimension $d \geq 2$ is an arbitrary constant in this thesis. Whenever an algorithm demands for a special k, I write ($k = 3$)-spacetree, e.g.

With a fixed Ω, each hypercube of the k-spacetree is either inside the computational domain or not. The k-section hence yields a spatial approximation of the computational domain as the surface of the inner hypercubes' union approximates the boundary of Ω. The k-section also yields a discretisation of this approximation, i.e. the k-spacetree gives an Ω_{h} (Figure 2.2).

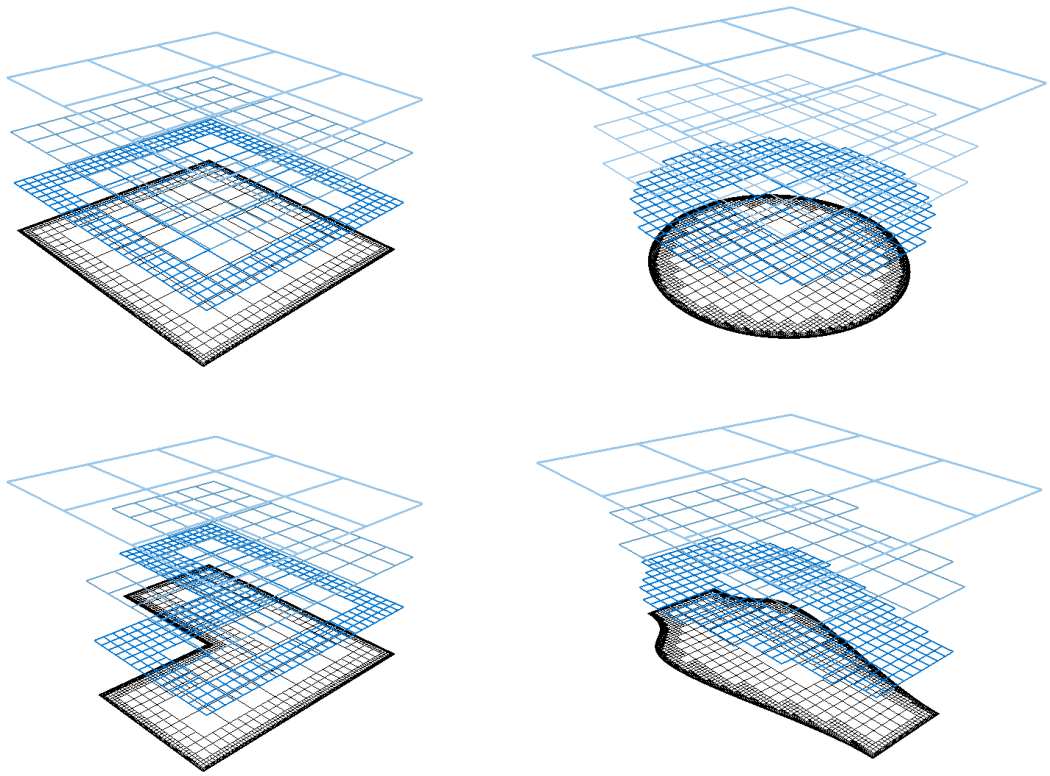


Figure 2.2: k-spacetree construction scheme for four different computational domains ($d = 2$, $k = 3$). Each illustration consists of four layers top-down: The layers one, two and three correspond to the three initial recursion steps, i.e. they show the spacetrees of height one, two and three. The bottom level shows a significant finer resolution of the domain.

2.2 Adaptive Cartesian Grids and k-spacetrees

Among the simplest grids in computational sciences is the Cartesian grid ([52], e.g.) consisting of hypercuboids whose faces' normals are parallel to a coordinate axis of the Cartesian coordinate system. The Cartesian grid's simplicity and structuredness make many geometric parameterisations and transformations in mathematical expressions and algorithms trivial or obsolete. The same two properties also enable for implementations exhibiting high performance and low memory requirements on today's hardware (e.g. [6, 22, 77])[4]. This section emphasises the obvious similarities of k-spacetrees and particular Cartesian grids, albeit k-spacetrees afford more flexible and more sophisticated grids and exhibit inherent grid hierarchy relation-

[4]Their poor $\mathcal{O}(h)$ approximation of complex geometries is picked up, discussed, and (at least) softened in Section 2.7.

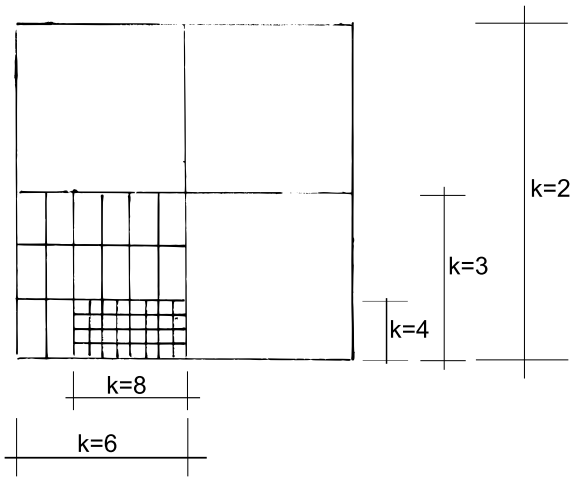


Figure 2.3: Adaptive Cartesian grid. This grid does not correspond to a k-spacetree, as k is not invariant throughout the construction, and it is not equal along the spatial directions.

ships. While the section recapitulates well-known facts familiar to most readers, subsequent chapters exploit both the similarities and the flexibility.

A *regular Cartesian grid* is a Cartesian grid with equally sized non-overlapping hypercuboids. Its construction resembles the recursive k-section in Section 2.1, but it refrains from the recursion and replaces the original cuboid. A straightforward extension of regular Cartesian grids is a *non-overlapping adaptive Cartesian grid*[5]. The corresponding grid generation process starts with a regular Cartesian grid. If appropriate, it then replaces each hypercube with another Cartesian grid and continues recursively.

The leaves of the k-spacetree yield a spatial discretisation of the unit hypercube. This tessellation is the *fine grid*. The term does not incorporate the actual shape of Ω. It does not hold any geometric information. It finally does thus not give a spatial discretisation Ω_h of Ω according to Section 2.1. The elements of Ω_h instead are a subset of the fine grid. I examine the interplay of a spacetree's fine grid and the geometry in Section 2.7. This interplay results in a fine grid for the computational domain instead of a fine grid for the spacetree's root.

Albeit the leaves and the fine grid interfere, a k-spacetree holds information beyond the pure adaptive mesh:

Example 2.1. *Let an equidistant Cartesian grid on a square with* 8×8 *geometric elements be the subject of inspection. Such a grid also corresponds to the fine grid of a* $(k = 8)$*-spacetree with height one. It furthermore corresponds to the fine grid of a* $(k = 2)$*-spacetree with height three. Although all three approaches yield exactly*

[5]Since all grids in this thesis are non-overlapping, I skip this qualifier from now on.

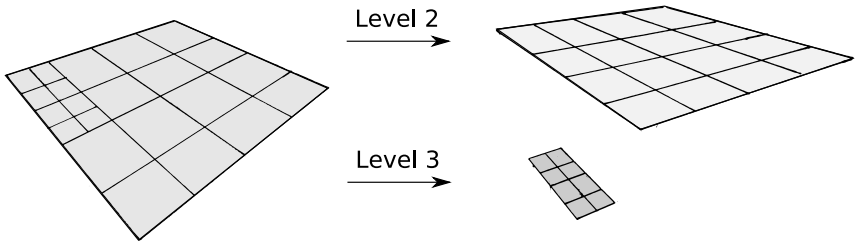


Figure 2.4: Fine grid of a two-dimensional ($k = 2$)-spacetree (left) and two Cartesian grids extracted from the tree (right).

the same spatial discretisation, each of them exhibits a topology of its own and a different number of geometric elements in total—$8 \cdot 8$ *for the Cartesian grid, and for the spacetrees* $8^2 + 1$ *or* $8^2 + 4^2 + 2^2 + 1$.

While each k-spacetree's fine grid equals an adaptive Cartesian grid, this does not hold the other way round, as most adaptive Cartesian grids choose the number of cuts along every hyperplane individually for each recursion step. These adaptive Cartesian grids thus are less restrictive (Figure 2.3). On the other hand, such an approach lacks the uniform hierarchy relations. A k-spacetree explicitly preserves the hierarchy, and the father-child relations' cardinalities are fixed and invariant. This facilitates many efficient and elegant algorithms—geometric multigrid solvers for example profit from the hierarchy.

In a k–spacetree, each hypercube has a *level*. If the construction needs at least ℓ recursion steps to create the hypercube, ℓ is the level. The root element thus has level zero. All the hypercubes of a given level in a k-spacetree have the same size. Yet, they do not define a Cartesian grid without any additional assumptions, as they might not cover the whole domain, as their layout might not be a hypercube itself, and as they might not be connected (Figure 2.4). In turn, each level of a full k–spacetree—all geometric elements not belonging to the maximum level are refined—really yields a regular Cartesian grid.

2.3 Some Formalism

The preceding sections introduce the spatial discretisation underlying this thesis' algorithms. Yet, they lack a rigorous formalism. Algorithms however demand for a finite, formal description, and they—by definition—have to be executable by a machine [11, 12]. The following pages establish such a formalism, and I also use them as opportunity to switch from a geometry's tessellation point of view to a grid-centered language including vertices and vertex-element adjacency relationships. New aspects beyond the discretisation are shifted to the subsequent Section 2.4, whereas Appendix C holds, beyond other details, a table of the definitions here.

The spatial discretisation Ω_{h} consists of a set $\mathbb{E}_{\mathcal{T}}$ of *geometric elements*. All geometric elements $e \in \mathbb{E}_{\mathcal{T}}$ are hypercubes. Thus, every geometric element has 2^d *vertices* $v_1, v_2, \ldots, v_d \in \mathbb{V}_{\mathcal{T}}$. All normals of the the $2d$ hypercubes' hyperfaces are parallel to a coordinate axis of the Cartesian coordinate system.

A k-spacetree $\mathcal{T}$ equals a four-tuple

$$\mathcal{T} = (\mathbb{E}_{\mathcal{T}}, \sqsubseteq_{\text{child}} \in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}, e_0 \in \mathbb{E}_{\mathcal{T}}, \mathbb{V}_{\mathcal{T}})$$

with a dedicated root element e_0 and a partial order $\sqsubseteq_{\text{child}}$ giving the child-father relationship. An element is a leaf, i.e. it belongs to the fine grid level, if it has no children. Elements with children, i.e. elements e with $\{e_i \in \mathbb{E}_{\mathcal{T}} : e_i \sqsubseteq_{\text{child}} e\} \neq \emptyset$, are *refined*, and $|\{e_i \in \mathbb{E}_{\mathcal{T}} : e_i \sqsubseteq_{\text{child}} e\}| = k^d$.

$$\begin{aligned} \mathcal{P}_{\text{refined}} : \mathbb{E}_{\mathcal{T}} &\mapsto \{\top, \bot\}, \\ \mathcal{P}_{\text{refined}}(e) &= \begin{cases} \top & \text{if} \quad \{e_i \in \mathbb{E}_{\mathcal{T}} : e_i \sqsubseteq_{\text{child}} e\} \neq \emptyset \\ \bot & \text{else} \end{cases}. \end{aligned}$$

Let

$$\begin{aligned} & & level : \mathbb{E}_{\mathcal{T}} &\mapsto \mathbb{N}_0 \quad \text{with} & \\ & & level(e_0) &= 0 \quad \text{and} & (2.1) \\ \forall e_i, e_j \in \mathbb{E}_{\mathcal{T}}, e_i \sqsubseteq_{\text{child}} e_j : & & level(e_i) &= level(e_j) + 1 & (2.2) \end{aligned}$$

return the level of a geometric element. The level of the root element equals zero (2.1), and the level of a child equals the parent's level incremented by one (2.2). The following properties result from the k-spacetree's construction algorithm:

$$\begin{aligned} \forall e_i, e_j \in \mathbb{E}_{\mathcal{T}}, e_i \neq e_j, level(e_i) = level(e_j) : & \quad |e_i \cap e_j| = 0, & (2.3) \\ \forall e_i, e_j \in \mathbb{E}_{\mathcal{T}}, e_i \sqsubseteq_{\text{child}} e_j : & \quad e_i \subset e_j, \quad \text{and} & (2.4) \\ \forall e_i \in \mathbb{E}_{\mathcal{T}}, \mathcal{P}_{\text{refined}}(e_i) : & \quad \bigcup_{e_j \sqsubseteq_{\text{child}} e_i} e_j = e_i, & (2.5) \end{aligned}$$

Different elements belonging to the same level share at most a submanifold. For $d = 2$ a vertex or an edge (2.3). Each element besides the root is contained within its father (2.4), and if all the children of one element are merged, the merged volume equals the father (2.5).

$vertex(e)$ with $vertex : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{V}_{\mathcal{T}}^{2^d} \subset \mathcal{P}(\mathbb{V}_{\mathcal{T}})$ identifies the 2^d adjacent vertices of an element e[6]. A vertex is a $(d+1)$-tuple $(x \in \mathbb{R}^d, level \in \mathbb{N}_0)$ holding the vertex's

[6]Because of the exponent 2^d, each image element has cardinality 2^d.

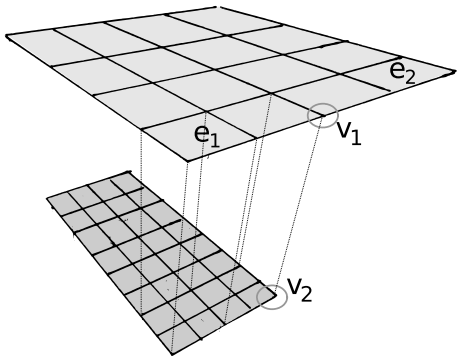


Figure 2.5: Part of a $(k = 2)$-spacetree with $d = 2$. $\mathcal{P}_{\text{refined}}(e_1)$ but $\neg\mathcal{P}_{\text{refined}}(e_2)$. e_1 has k^d children. v_1 and v_2 are at the same position in space but belong to different levels. Thus, they are not equal. $v_1 \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$. $v_2 \in \mathbb{H}_{\mathcal{T}}$ is a hanging node.

level and position in space. Two vertices are equal if and only if they coincide in both space and level. The level $level(v)$ of a vertex v of an element equals the element's level:

$$\forall v \in \mathbb{V}_{\mathcal{T}}, \forall e \in adjacent(v) : level(e) = level(v).$$

The set

$$\mathbb{V}_{\mathcal{T}} = \bigcup_{e \in \mathbb{E}_{\mathcal{T}}} vertex(e)$$

holds all vertices of the spacetree. This definition derives vertices from elements. The counterpart

$$adjacent : \mathbb{V}_{\mathcal{T}} \mapsto \mathcal{P}(\mathbb{E}_{\mathcal{T}})$$

delivers all the elements that are adjacent to a vertex on the same level. The spacetree's construction yields

$$\forall v \in \mathbb{V}_{\mathcal{T}} : 0 < |adjacent(v)| \leq 2^d,$$

and this adjacency information allows for a classification of the vertices: A vertex v with less than 2^d adjacent elements, i.e. $|adjacent(v)| < 2^d$, is a *hanging node* or *hanging vertex*. To distinguish hanging vertices from the other vertices is essential throughout the thesis, and the set

$$\mathbb{H}_{\mathcal{T}} = \{v \in \mathbb{V}_{\mathcal{T}} : |adjacent(v)| < 2^d\} \tag{2.6}$$

provides this separation.

The following convention is common for vertices in many applications such as geometric ansatzspaces in Chapter 4, where it ensures the continuity of the numerical solution: Hanging vertices never hold information. As a result, there is no need to store hanging nodes persistently.

2.4 Element-wise Grid Traversal

On the next pages, I derive a grid traversal idea fitting to both an element-wise traversal of the adaptive Cartesian grid and the k-spacetree. This traversal is the basis of all algorithms implemented later on. In this context, the conclusions and restrictions coming along with it (Definition 2.1) particularly coin the thesis.

A function on a k-spacetree maps the tree and data assigned to it to an image. Iterative solvers, e.g., map the data structure to another k-spacetree; plotters, e.g., write it to an output stream whose content is interpreted by a visualisation tool. Without any assumptions on the preimage's impact, such algorithms have to read each element of the spacetree at least once, i.e. they have to process each element of $\mathbb{E}_\mathcal{T}$ and each element of $\mathbb{V}_\mathcal{T}$. This is a *grid traversal*. The interplay of geometric elements and vertices is essential in most algorithms. A traversal algorithm having minimal computational complexity reads each vertex and each geometric element once. If the connectivity information is of interest, the traversal algorithm has to read each vertex-element relationship at least once.

Two different traversal types are *vertex-wise* and *element-wise*. Vertex-wise traversals define a total order on the vertices and run through this data stream. For each vertex v, they also process the adjacent elements $e \in adjacent(v)$. Thus, each vertex is read once, but elements are read multiple times. Element-wise algorithms in contrast define a total order on the geometric elements and run through this data stream. For each element e, they also process the adjacent vertices $v \in vertex(e)$. Thus, each element is read once, but vertices are read multiple times: for $v \in \mathbb{V}_\mathcal{T} \backslash \mathbb{H}_\mathcal{T}$ exactly 2^d. For hanging nodes $v \in \mathbb{H}_\mathcal{T}$ the number of reads is smaller. In this thesis, all algorithms are based upon an element-wise traversal[7].

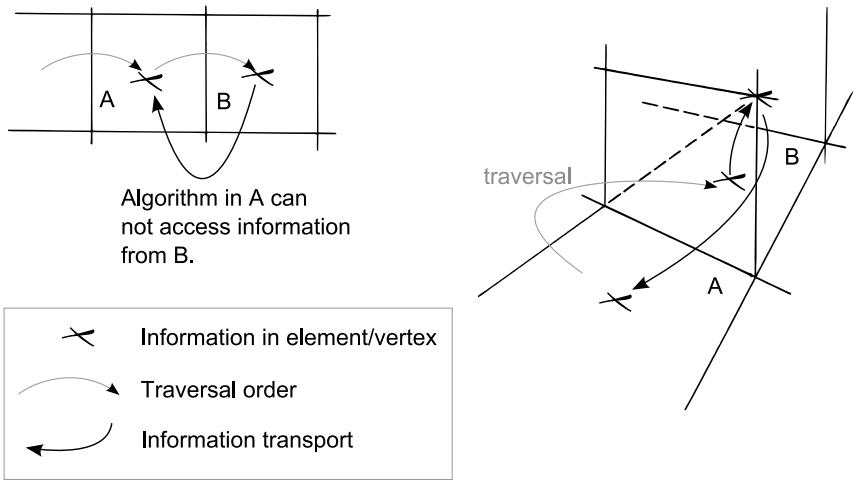
Definition 2.1. *An* element-wise k-spacetree traversal *processes the k-spacetree $\mathcal{T}$ in $n \geq |\mathbb{E}_\mathcal{T}|$ steps. In each step, exactly one element $e \in \mathbb{E}_\mathcal{T}$ and the 2^d adjacent vertices $v \in adjacent(e)$ are available to an algorithm built atop the traversal. In-between two steps, both the source and the destination data are available.*

The element-wise k-spacetree traversal poses an important restriction on the speed information is transported from one geometric element to another: Without further assumptions, it is not possible for an element-wise algorithm to directly evaluate or

[7] Both vertex-wise and element-wise traversal are found in scientific computing. Some examples: Introductions to the finite element method usually start from a vertex point of view ([8], e.g.). Finite element codes switch to an element-wise traversal often to eliminate duplicate computations such as numerical integration on the elements. Grid-based visualisation software typically prefers a vertex-wise point of view, whereas an element-wise traversal fits better into the paradigm of volume rendering. For spacetrees, algorithms stemming from graph theory prefer an element-wise approach as it fits to standard tree search algorithms. Other space tree formalisms such as Morton ordering [57] correspond to a vertex-wise interpretation of the grid.

even manipulate neighbour cells, as at most two neighbours' data is available during an element transition.

Example 2.2. *The element-wise traversal grants an algorithm access to the current element and its vertices. The two sketches below illustrate the resulting challenges for an element A traversed before an element B:*



If $traversal(A) < traversal(B)$ and if an algorithm requires information from B within A, this information is not available (left). Thus, information needed by neighbours has to be written to the element's vertices (B right). This information then is available to the neighbours in the subsequent iteration (A right).

If a cell needs information from neighbouring cells, there has to be a function that reconstructs the neighbour's information. Such a function analyses the properties of the vertices that are adjacent to both geometric elements and it uses the following information:

1. For the reconstruction of the neighbours sharing a common hyperface, the 2^{d-1} shared vertices and the elements' data itself have to be sufficient.

2. For the reconstruction of the neighbours sharing a common hyperedge, the element's state, the 2^{d-2} shared vertices and the information obtained in the preliminary step have to be sufficient.

...

d. For the reconstruction of the neighbours sharing only one common vertex, the element's state, the 2^0 common vertices, i.e. the only common vertex, and the information obtained in the preliminary steps have to be sufficient.

Vertices act as an information transport medium: Element-based information propagates due to the vertices (write from origin element to vertex; image element then takes data from this vertex). Vertex-based information propagates from one vertex to the neighbouring vertex. The maximum information propagation speed the element-wise traversal guarantees without further assumptions and knowledge on the elements' and vertices' order and topology thus equals on element per traversal.

For any algorithm mapping a k-spacetree's data to another k-spacetree, a computer scientist is interested if it is possible to use the same data structure as origin and as destination record. If this is possible, the application's demand for memory halves. The answer depends on the algorithm's vertex manipulation and the reconstruction function built atop.

If an algorithm needs, at any time, a neighbour element's state in the preimage, and if it manipulates the current traversal element's state, it requires the state of a vertex to remain invariant throughout the traversal. Yet, each element's state transition implies an update of the vertices, as the vertices transport information. Thus, it is not possible to make the input data structure, i.e. the preimage's vertices, act as output data structure.

Instead of using two complete data structures, any algorithm harming this criterion can be implemented using protocol variables: A vertex's property transition is not performed immediately, but the algorithm writes the transition into a transaction variable. At the end of the traversal, all transactions are processed en bloc. Alternatively, the very last write operation on a vertex precedes the transaction. Such approaches are well-known from databases and persistence layers ([24], e.g.).

If a vertex's state changes exclusively in-between two iterations, there is a maximum guaranteed speed for the information to spread. For a fixed grid level, information can spread only one element per iteration: In one iteration, an element e_1 writes its information to the vertices. The updated information is available to the neighbouring elements e_2 with $traversal(e_2) < traversal(e_1)$ in the subsequent traversal. The other way round, this is not a restriction similar to a CFL condition [8] for the whole spacetree, i.e. it does not imply that information can only be passed from one element to a neighbouring element per iteration. If an algorithm traverses from an geometric element e_1 to its father element e_2, then to e_2's neighbour e_3, and then to a child e_4 of cell e_3, there is no need for e_1 and e_4 to be neighbours, i.e. information stemming from element e_1 can influence a non-neighbouring element e_4.

2.5 k-spacetree Traversals

The preceding section discusses element-wise traversals from a grid point of view. It is an obvious idea to transfer the ideas to the tree world. A unified traversal concept

holding for both point of views allows each algorithm to pick out the formalism leading to the simplest description, while Peano's algorithmic realisations always benefit from the tree paradigm.

A *tree traversal* in graph theory is a process that reads each node of a tree once [47]. Each tree node in a k-spacetree equals a geometric element in the hierarchical grid. Thus, a tree traversal yields an element-wise traversal according to Definition 2.1, if one makes the tree traversal algorithm also pass the corresponding grid vertices.

Several different criterion induce a classification of tree traversals. The distinction of *deterministic* and *nondeterministic* traversals is of importance in this thesis: Parallel algorithms for example exhibit nondeterministic behaviour, i.e. sophisticated synchronisation mechanisms are required if the code's correctness demands for deterministic algorithmic steps. The hierarchy's influence on the traversal order also is essential for most algorithms:

Definition 2.2. *A traversal preserves the child-father relationship* $\sqsubseteq_{\text{child}}$*, if*

$$e_1 \sqsubseteq_{\text{child}} e_2 \Rightarrow traversal(e_2) < traversal(e_1).$$

A traversal preserves the inverse child-father relationship $\sqsubseteq_{\text{child}}$*, if*

$$e_1 \sqsubseteq_{\text{child}} e_2 \Rightarrow traversal(e_1) < traversal(e_2).$$

With binary trees, the preorder traversal [11, 12] preserves the child-father relationship's implication. With trees such as k-spacetrees, where a refined element has more than two children, the standard *depth-first* and the *breadth-first* orders' $traversal$ functions also fit to the first definition above [11, 12]. Hereby, Definition 2.2 neither determines whether the traversal is deterministic, nor do the implications prefer a depth-first or a breadth-first traversal.

Let an algorithm's traversal preserve both relationships. Such an algorithm can transfer information from fathers to their children within one traversal transition according to the child-father relationship. Information hereby is transported top-down. For the inverse child-father relationship, the algorithm can transfer all the children's information to their father within one iteration. Information is transported bottom-up. The first principle corresponds to information inheritance within tree algorithms. The second principle describes information analysis [46].

A bigger part of this thesis turns its attention to the efficient realisation of the grid and the grid traversal. Depth-first traversals (later merged with the paradigm of breadth-first) build the basis of these realisations, as their traversal management requires only one data structure—a stack or a queue (Algorithm 2.1 or Algorithm 2.2)[8]. The names and semantics of the operations and data structures in the pseudocode follow standard text books such as [11, 12]. A key ingredient for an efficient

[8]The actual realisation is a recursive code, i.e. the *stack* in Algorithm 2.1 is hidden by the call stack.

realisation is the transformation of the abstract, nondeterministic traversal definition into a deterministic sequence. The nondeterminism results from the **forall** statements in the pseudocode. Introducing an order for these statements makes the traversal deterministic.

Algorithm 2.1 Depth-first traversal.

$$\begin{aligned} \mathcal{T} &= (\mathbb{E}_{\mathcal{T}}, \sqsubseteq_{\text{child}} \in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}, e_0 \in \mathbb{E}_{\mathcal{T}}, \mathbb{V}_{\mathcal{T}}) \\ stack &:= (e_0) \qquad \text{when algorithm starts up} \end{aligned}$$

1: **procedure** $dfs(stack)$
2: $\quad e_{\text{current}} \leftarrow \textbf{pop}_{stack}()$
3: $\quad$ **for all** $e \in \mathbb{E}_{\mathcal{T}} : e \sqsubseteq_{\text{child}} e_{\text{current}}$ **do**
4: $\quad\quad \textbf{push}_{stack}(e)$
5: $\quad$ **end for**
6: **end procedure**

Algorithm 2.2 Breadth-first traversal.

$$\begin{aligned} \mathcal{T} &= (\mathbb{E}_{\mathcal{T}}, \sqsubseteq_{\text{child}} \in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}, e_0 \in \mathbb{E}_{\mathcal{T}}, \mathbb{V}_{\mathcal{T}}) \\ queue &:= (e_0) \qquad \text{when algorithm starts up} \end{aligned}$$

1: **procedure** $bfs(queue)$
2: $\quad e_{\text{current}} \leftarrow \textbf{dequeue}_{queue}()$
3: $\quad$ **for all** $e \in \mathbb{E}_{\mathcal{T}} : e \sqsubseteq_{\text{child}} e_{\text{current}}$ **do**
4: $\quad\quad \textbf{enqueue}_{queue}(e)$
5: $\quad$ **end for**
6: **end procedure**

From a hardware-near and implementation-aware point of view, a straightforward implementation of the two traversals exhibits a subtle disaccord with a naive reception of the term element-wise. Each refined geometric element in the spacetree is read twice: Once by the **pop** operation, once by the **push** (**queue** and **dequeue** respectively). And each operation has to move records within the memory and, thus, accesses the data. I weaken the phrase "element is read once" in a minute, and regard both traversals fit to the definition of element-wise. Herefrom, the depth-first traversal makes the following statements possible.

- Each element on the stack corresponds to an element in the adaptive Cartesian grid. Let the stack's top element determine the *current element* of the element-wise traversal.

- For each refined geometric element, the traversal triggers k^d **push-pop** combinations. They correspond to a grid traversal transition from an element on a given level ℓ into a subelement on a level $\ell + 1$—a *step-down* transition. Both the parent and the child data are available throughout this transition.
- Each **pop** operation corresponds to a transition from a geometric element into a sibling or back to the father element (*bottom-up transition*). Both source and destination element's data are available throughout the transition.

The latter transition agrees with the first issue, as the stack's top element determines the geometric element. Each refined element occurs twice within the sequence of traversed geometric elements, and this is a conflict with the radical interpretation of the term traversal. I resolve this conflict as I replace the *traversal* function by two access functions. Each of them alone defines a traversal in the traditional sense.

Definition 2.3. *Let* $first : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{N}_0$ *denote the first time an element is read. Let* $second : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{N}_0$ *denote the second time an element is read. Let* e_1 *be a descendant of* e_2 *or the other way round.* $first(e) = second(e) \Leftrightarrow \neg\mathcal{P}_{\text{refined}}(e)$*. All elements are read at most twice. A process with*

$$\begin{aligned} e_1 \sqsubseteq_{\text{child}} e_2 \Rightarrow first(e_2) &< first(e_1), \qquad \textit{and} \\ second(e_1) &< second(e_2) \end{aligned}$$

is a k-spacetree traversal.

$first$ hereby preserves the father-child relationships, *second* preserves the inverse child-father relationships. A k-spacetree traversal defines an element-wise traversal for all Cartesian grids of the k-spacetree, and the information access discussion from Section 2.4 is valid for it. All traversals in this thesis are k-spacetree traversals.

2.6 Vertex-based Refinement and Information Transport

The subsequent section discusses the k-spacetree construction process and how the spacetrees are encoded. Due to the information propagation property, the refinement information is assigned to the vertices, and an element in turn is refined if and only if at least one adjacent vertex holds a corresponding refinement flags. As a result, the algorithm deduces hanging nodes on-the-fly from the refinement predicate $\mathcal{P}_{\text{refined}}$ of coarser levels, i.e. no additional storage effort is spent on hanging vertices. Furthermore, it turns out that the time to set the refinement predicate has to be chosen carefully, if the number of read operations for vertices $v \notin \mathbb{H}_{\mathcal{T}}$ has to be invariant. This property is essential for many algorithms relying on the fact that

all elements adjacent to a vertex are traversed. In the following, I introduce this refinement flag for the vertices and derive both the elements' refinement and the term hanging node from this predicate.

For many element-wise algorithms in this thesis, it is essential to be able to decide whether a vertex $v \in \mathbb{H}_{\mathcal{T}}$. According to (2.6), a vertex is a hanging vertex if the number of adjacent elements is smaller than 2^d. As the number of its adjacent elements depends on the refinement predicates on the level above, the vertex's state, i.e. whether $v \in \mathbb{H}_{\mathcal{T}}$ or not, depends on the level above.

Example 2.3. *Let $v_l = (x, l)$ and $v_{l+1} = (x, l+1)$ be two vertices at the same position $x \in \mathbb{R}^d$ in space. The number of adjacent elements of v_{l+1} depends on the state of the adjacent elements of v_l—if one adjacent element of v_l is not refined, the number of adjacent elements of v_{l+1} is smaller than 2^d and v_{l+1} is hanging.*

Let e_l and e_{l+1} be two elements with $e_{l+1} \sqsubseteq_{\text{child}} e_l$ and $v_l \in vertex(e_l), v_{l+1} \in vertex(e_{l+1})$. Throughout the top-down transition from e_l to e_{l+1}, the refinement state of e_l is available. The refinement states of the other elements that are adjacent to v_l are not available. To bridge that information gap, v_l carries a variable reconstructing the state of all the adjacent elements. It makes no sense to hold refinement information redundantly. Thus, exclusively the vertices hold refinement information.

Definition 2.4. *The k-spacetrees in this thesis are based upon an* or-wise vertex refinement criterion*. There's a refinement predicate $\mathcal{P}_{\text{refined}}$ for each vertex with*

$$\forall e \in \mathbb{E}_{\mathcal{T}} : \mathcal{P}_{\text{refined}}(e) \Leftrightarrow \exists v \in vertex(e) : \quad \mathcal{P}_{\text{refined}}(v), \quad \textit{and} \quad \forall v \in \mathbb{H}_{\mathcal{T}} : \quad \neg\mathcal{P}_{\text{refined}}(v), \tag{2.7}$$

i.e. every time the refinement predicate holds for a vertex, all the 2^d adjacent geometric elements are refined.

For many algorithms it is extremely useful to ensure that the algorithm processes non-hanging vertices exactly 2^d times. As one can distinguish $first$ and *second* transitions, the term "processes" refers to one of these orders. The next theorem formalises this constraint and relates it to the information propagation speed.

Theorem 2.1. *Consider any dfs-type k-spacetree traversal. Any vertex $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ is read exactly 2^d times, i.e. the traversal accesses every adjacent geometric element of a non-hanging vertex, if and only if the refinement predicate $refine(v)$ does not change throughout the traversal.*

Proof. If the grid does not change throughout the traversal, each vertex is processed 2^d times, as the k-spacetree traversal processes all elements of the k-spacetree once. To show that this does not hold anymore if the grid changes, it is sufficient to give one counterexample (see Figure 2.7). □

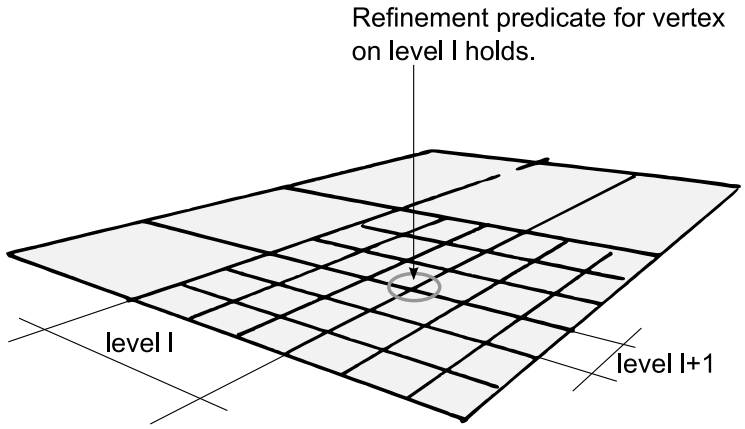


Figure 2.6: If $\mathcal{P}_{\text{refined}}$ holds for a vertex, all the adjacent geometric elements are refined. Within one element, all the vertices' refinement flags are combined via an logical or to determine whether the element itself is refined (Figure 2.6).

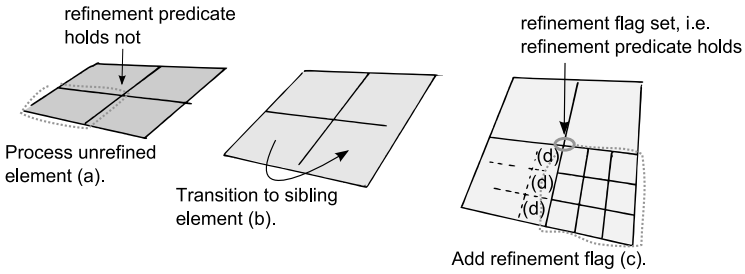


Figure 2.7: Motivation for Theorem 2.1: (a) The k-spacetree traversal processes an unrefined element e_a. (b) It continues to sibling element e_b. (c) Within e_b, the algorithm sets the refinement flag of an adjacent vertex. The k-spacetree traversal will continue with the new elements $e \sqsubseteq_{\text{child}} e_b$, but (d) it will never descend into the new elements $e \sqsubseteq_{\text{child}} e_a$.

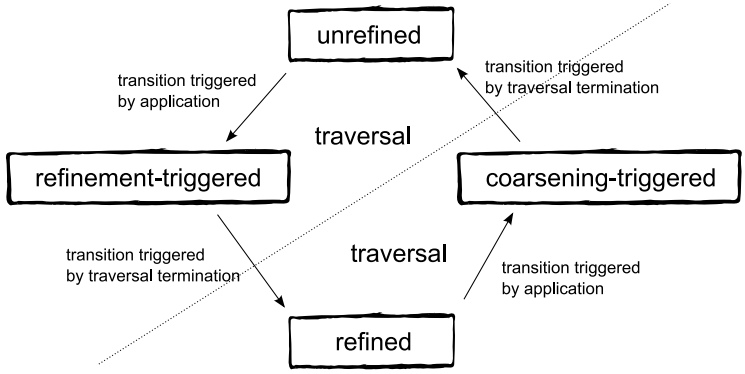


Figure 2.8: The vertex's refinement state is synchronised with the traversal: Any algorithm may trigger a refinement at any time. The refinement triggered state is changed into refined in-between two iterations. The grid structure thus is invariant throughout the traversal.

With the or-wise vertex refinement criterion and a fixed number of vertex accesses, a vertex's refinement state has to remain invariant throughout the traversal. As many algorithms want to change the grid's layout whenever they "feel" like that, the vertices' carry multiple refinement states with a transaction semantics. If an algorithm wants to refine a vertex, it switches the vertex's state from `unrefined` to `refinement-triggered`. The refine predicate does not hold for the additional state `refinement-triggered`. At the end of the traversal, the algorithm takes all the vertices with state `refinement-triggered` and switches the vertex's state from `refinement-triggered` to `refined`. The refinement predicate does hold for `refined`. Instead of a postprocessing of all vertices, the transition is implemented after a vertex is read the last time. An analogous argumentation is valid for the coarsening of refined vertices, i.e. for algorithms that change a vertex's state from `refined` to `unrefined`, and the underlying state chart is given in Figure 2.8. The outlined solution with a transaction variable is only one possible solution to this problem. If setting the refinement predicate is allowed only at the first read, no transaction variable would be required. I though preferred to make the refinement criterion and decision algorithms completely encapsulated from the grid management.

Within an efficient implementation, the exist quantifier in (2.7) has to be resolved. A brute force search for all neighbour elements is ill-suited. There is an elegant formulation based upon the definition of the father vertices (Algorithm 2.3). This formulation accepts the parent's vertices in cell-wise lexicographic order and the position of the vertex within the refined parent element. It then derives the state of an element's vertex from these inputs.

Theorem 2.2. *With Algorithm 2.3, a non-hanging node is given by*

$$v_i \in \mathbb{E}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} \Leftrightarrow \bigvee_{v \in father(V_{parent}, i)} \mathcal{P}_{\text{refined}}(v).$$

V_{parent} denotes the 2^d vertices of the father cell. $\bigvee$ denotes a logical or over all the functions' results and motivates the term or-wise refinement criterion.

Proof. An induction over the dimension d proves the theorem for any k.

- With $d = 1$, the father element is a line split up into k subelements.
  - For $number = 0$, $father(V, 0)$ returns the very left vertex of the refined element. If the father's left vertex is not refined, the father's left neighbour element is not refined, either. v_0 is a hanging vertex.
  - For $number = k$, $father(V, k)$ returns the very right vertex of the refined element. If the father's right vertex is not refined, the father's right neighbour element is not refined, either. v_k is a hanging vertex.
  - For $number \in \{1, \ldots, k-1\}$, both vertices adjacent to the coarse refined element are contained within $father(V, number)$'s result set. One of them holds the refinement predicate as the element is refined. v_{number} thus has two adjacent geometric elements and is not a hanging vertex.

- With $d \mapsto d + 1$, the proof has to distinguish $v_{d+1} \in \{1, \ldots, k-1\}$ from $v_{d+1} \in \{0, k\}$:
  - If $v_{d+1} \in \{0, k\}$, the vertex v belongs to a coarse element face with a normal parallel to the x_d-axis. *parentno*'s corresponding entry is set to this face's coordinate, i.e. 0 for the face nearer to the coordinate system's origin or 1 respectively. The decision and analysis of this face's vertices then reduces to a d-dimensional challenge.
  - If $v_{d+1} \in \{1, \ldots, k-1\}$, the vertex v does not belong to a face with a normal parallel to the x_{d+1}-axis. If the direct neighbour vertices along the x_{d+1}-axis are not hanging, the vertex itself is not hanging, too. As this arguing is to be repeated, the next two neighbours along the x_{d+1}-axis that are element of the coarse element's face with normal $d+1$ determine whether the vertex is hanging. If one of them is not hanging, the vertex also is not hanging: it then has 2^{d+1} adjacent geometric elements. The problem reduces to two d-dimensional challenges.

□

Algorithm 2.3 Derive the father vertices, i.e. the vertices of the refined father element that influence a vertex. Algorithm accepts 2^d vertices adjacent to the father element and the vertex's position within k^d refinement patch.

$$\begin{aligned} father : \left(\mathcal{P}(\mathbb{V}_{\mathcal{T}}) \times \{0,\ldots,k-1\}^d\right) &\mapsto \mathcal{P}(\mathbb{V}_{\mathcal{T}}) \qquad \text{with} \\ father(V_{parent}, number) &= V'_{parent}, \\ |V_{parent}| &= 2^d \qquad \text{and} \\ V'_{parent} &\subset V_{parent}. \end{aligned}$$

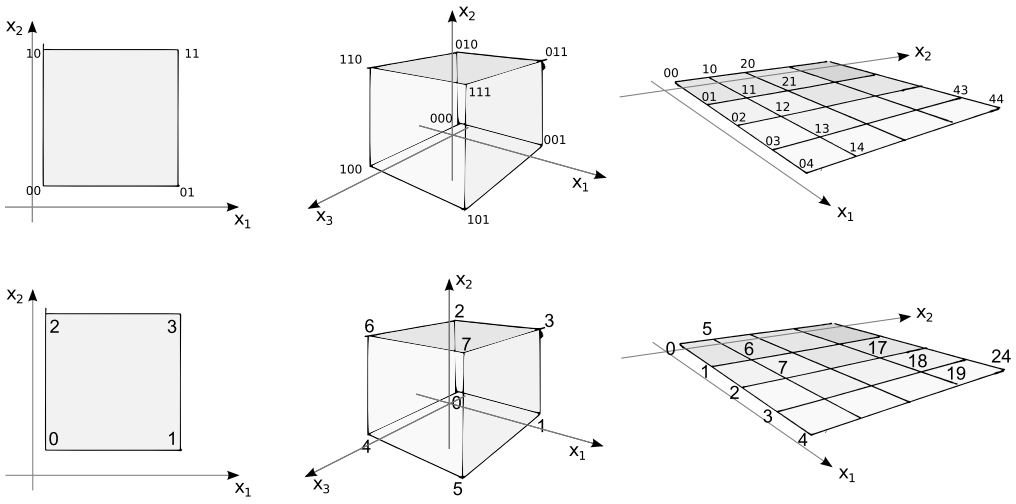
```
1: procedure father(V_parent, number)
2:     V'_parent ← ∅                                          ▷ Result set.
3:     parentno ← (0, ..., 0)
4:     for i ∈ {0, d − 1} do
5:         if number_i = 0 then
6:             parentno_i ← 0
7:                                  ▷ number and parentno are d-tuple.
8:                                  ▷ Index _i denotes the ith entry, C-style.
9:         else if number_i = k then
10:            parentno_i ← 1
11:        else
12:            newnumber1, newnumber2 ← number
13:            newnumber1_i ← 0
14:            newnumber2_i ← k
15:            V'_parent ← V'_parent ∪ father(V_parent, newnumber1)   ▷ Recursive call.
16:            V'_parent ← V'_parent ∪ father(V_parent, newnumber2)   ▷ Recursive call.
17:            return V'_parent
18:        end if
19:    end for
20:    V'_parent ← {v ∈ V_parent : number(v) = parentno}
21:                ▷ Result contains vertex at same position in the level above
22:    return V'_parent                                       ▷ Recursion terminates
23: end procedure
```

—Lexicographic Enumeration—

The *lexicographic enumeration* defines an enumeration for all vertices of one cell or all vertices of one refinement step. It assigns each vertex a d-tuple

$$\begin{aligned} number : v \in \mathbb{V}_{\mathcal{T}} &\mapsto \{0, \ldots, k-1\}^d \qquad \text{and a} \\ number_{linearised} : v \in \mathbb{V}_{\mathcal{T}} &\mapsto \mathbb{N}_0 \qquad \text{with} \\ number_{linearised} = \sum_{i=0}^{d-1} number_i(v) \cdot k^i. & \end{aligned}$$

Within a single cell, $k = 1$ gives a vertex enumeration. Otherwise k equals the k in k-spacetree and *number* is the vertex's position within a k^d patch, i.e. within one refined geometric element.



Within a k^d motif, the enumeration starts with the vertex nearest to the origin of the Cartesian coordinate system. It then enumerates all the vertices along the x_1-axis. Afterwards, it continues with the x_2-axis, etc. In the two-dimensional case, the enumeration starts with the left bottom vertex, and it enumerates from left to right and bottom-up.
Instead of working with the function $number(v)$, this thesis often denotes the index as subscript, i.e. $v_i \Leftrightarrow number(v_i) = i$. If vertices of different levels are involved, two subscripts are used. The first denotes the level, the second the index.

2.7 Geometry Representation

Cartesian grids are not aligned with the computational domain, i.e. their cells cover parts outside and inside the domain. Since a PDE is defined on a computational domain, the domain's shape information has to be transferred to the spatial discretisation. Afterwards, one can solve the PDE numerically. The mapping from the continuous domain to the k-spacetree is discussed by this section emphasising the identification of vertices that are inside the computational domain or on the domain's boundary. For the boundary vertices, the PDE's boundary conditions determine the vertices' state: there are inner vertices where the numerics determine the solution, there are boundary vertices with prescribed properties, and there are ignored outer vertices. At the end of the section, I present a simple geometric refinement criterion.

The spacetree construction already involves geometric information since the root element covers the computational domain. It is not inside but covers the domain's boundary. I continue this distinction into inner, outer and hybrid elements for each k-spacetree element and end up with a *marker-and-cell approach*—[38] synonymously refer to it as "marker and cell technique". The union of the inner cells then gives the computational domain. As this approach does not approximate the computational domain exactly, it demands for a mapping of the computational domain's boundary to the boundary points of the Cartesian grids, and it demands for a discussion of the accuracy.

Definition 2.5. *A geometric element $e \in \mathbb{E}_{\mathcal{T}}$ is* inside *the computational domain if $e \subseteq \overline{\Omega}$. Ω is open, e is closed, $\overline{\Omega}$ is the closure of Ω. $\mathcal{P}_{\text{inside}}(e)$ and $\neg\mathcal{P}_{\text{outside}}(e)$ hold. Otherwise, it is* outside*, i.e. $\neg\mathcal{P}_{\text{inside}}(e)$ and $\mathcal{P}_{\text{outside}}(e)$.*

Definition 2.6. *A vertex $v \in \mathbb{V}_{\mathcal{T}}$ is*

- outside *the computational domain, if all 2^d adjacent elements $e \in adjacent(v)$ are outside. $\mathcal{P}_{\text{outside}}(v)$ holds.*
- a boundary vertex*, if $1 \leq k < 2^d - 1$ adjacent elements $e \in adjacent(v)$ are outside. $\mathcal{P}_{\text{boundary}}(v)$ holds.*
- inside *the computational domain otherwise, i.e. if all 2^d adjacent elements $e \in adjacent(v)$ are inside. $\mathcal{P}_{\text{inside}}(v)$ holds.*

These two classifications lead to a number of interesting statements on the approximation of the continuous computational domain. All the statements rest upon the property that the computational domain is shrinked towards the smaller computational domain. Within the multiscale context, this proves of great value, although alternative formulations of the inside/outside/boundary predicates are possible and might make sense in another context.

PDEs are defined on the computational domain. Algorithms to solve a PDE on a grid, algorithms to visualise a solution, and so forth thus perform a number of fixed operations on elements and vertices inside or at the boundary of the discretised computational domain. There's nothing to compute on elements outside the computational domain. Thus, it is reasonable to make the traversal check each element's state before it triggers any user-defined operation. As k-spacetree traversals preserve the child-father relationship, a check exhibiting that a geometric element is inside the computational domain makes all the subsequent checks for the descendants' state obsolete. For such an code optimisation the next property is useful.

Corollary 2.1.

$$\begin{aligned} \mathcal{P}_{\text{inside}}(e) &\Leftrightarrow \forall e_i \sqsubseteq_{\text{child}} e : \mathcal{P}_{\text{inside}}(e_i) \qquad \textit{and} \\ \forall e_i \sqsubseteq_{\text{child}} e : \mathcal{P}_{\text{outside}}(e_i) &\Rightarrow \mathcal{P}_{\text{outside}}(e). \end{aligned}$$

Proof. Both statements result directly from the fact that the union of all children of a parent equals the parent's geometric element.

$$\begin{aligned} \bigcup_{e_i \sqsubseteq_{\text{child}} e} e_i &= e, \\ \bigcup_{e_i \sqsubseteq_{\text{child}} e} e_i \subseteq \overline{\Omega} &\Leftrightarrow e \subseteq \overline{\Omega} \Leftrightarrow \mathcal{P}_{\text{inside}}(e) \qquad \text{and} \\ \exists e_i, e_i \sqsubseteq_{\text{child}} e : \mathcal{P}_{\text{outside}}(e_i) &\Rightarrow \exists e_i, e_i \sqsubseteq_{\text{child}} e : e_i \not\subseteq \overline{\Omega} \Rightarrow \bigcup_{e_i \sqsubseteq_{\text{child}} e} e_i \not\subseteq \overline{\Omega} \\ &\Rightarrow \neg\mathcal{P}_{\text{inside}}(e) \Leftrightarrow \mathcal{P}_{\text{outside}}(e). \end{aligned}$$

□

A k-spacetree yields *adaptive Cartesian* grids for a given level and arbitrary computational domain Ω. Let

$$\begin{aligned} \Omega_{\text{h},\ell} &= \{e \in \mathbb{E}_{\mathcal{T}} : level(e) = \ell \wedge \mathcal{P}_{\text{inside}}(e)\}, \qquad \text{and} \\ \Omega_{\text{h},\ell}^{\text{adaptive}} &= \{e \in \mathbb{E}_{\mathcal{T}} : \mathcal{P}_{\text{inside}}(e) \wedge (level(e) = \ell \vee level(e) < \ell \wedge \neg\mathcal{P}_{\text{refined}}(e))\}. \end{aligned}$$

Depending on the context, I also refer to $\Omega_{\text{h},l}$ as grid of the level ℓ. $\Omega_{\text{h},l}^{\text{adaptive}}$ defines the adaptive Cartesian grid with maximum level ℓ. The *computational fine grid* or *fine grid* of a k-spacetree is

$$\Omega_{\text{h}} = \{e \in \mathbb{E}_{\mathcal{T}} : \mathcal{P}_{\text{inside}}(e) \wedge \neg\mathcal{P}_{\text{refined}}(e)\}.$$

I use the same symbol for fine grid, spatial discretisation and computational fine grid, i.e. I do not distinguish between the spacetree's leaves and the fine grid restricted to the inner geometric elements, as the context makes the semantics unambiguous.

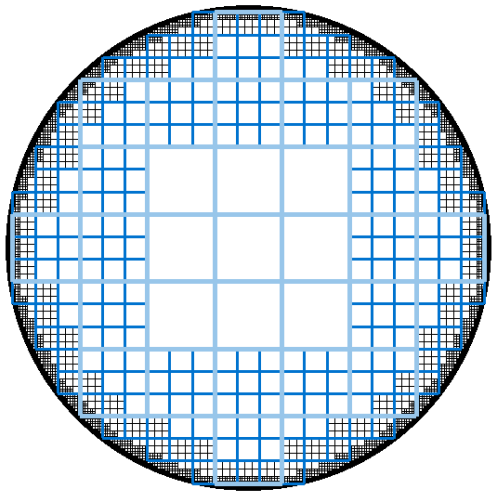


Figure 2.9: The bigger the maximum level of the refinement criterion in (2.9), the better the approximation of the computational domain becomes. Here, it is a circle. The domain's volume grows with increasing level, i.e. computational domains with a coarser resolution are contained within the discrete computational domain belonging to finer resolutions.

Theorem 2.3. *Let* $\sqsubseteq \in \mathcal{P}(\mathbb{E}_{\mathcal{T}}) \times \mathcal{P}(\mathbb{E}_{\mathcal{T}})$ *denote that elements of one discretisation tessellate another discretisation, i.e. each primitive of the set on the left-hand side fits completely into a primitive of the set on the right-hand side.*

$$\Omega_{\mathrm{h},1}^{\mathrm{adaptive}} \sqsubseteq \Omega_{\mathrm{h},2}^{\mathrm{adaptive}} \sqsubseteq \Omega_{\mathrm{h},3}^{\mathrm{adaptive}} \sqsubseteq \ldots \sqsubseteq \Omega_{\mathrm{h}}. \tag{2.8}$$

Proof. The statement $\Omega_{\mathrm{h},\ell}^{\mathrm{adaptive}} \sqsubseteq \Omega_{\mathrm{h},l+1}^{\mathrm{adaptive}}$ directly results from Corollary 2.1. The final $\sqsubseteq$ relation results from

$$\lim_{l \to \infty} \{e \in \mathbb{E}_{\mathcal{T}} : level(e) = \ell \vee level(e) < \ell \wedge \neg \mathcal{P}_{\mathrm{refined}}(e)\} = \{e \in \mathbb{E}_{\mathcal{T}} : \neg \mathcal{P}_{\mathrm{refined}}(e)\}.$$

□

The spatial approximations of the different levels of a k-spacetree converge monotonously to the exact computational domain (Theorem 2.3), and the finer the grid level, the better the approximation (Figure 2.9).

All this formalism affords the definition of a geometric adaptivity criterion. For all vertices $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ fulfilling

$$level(v) \leq \ell \wedge \mathcal{P}_{\mathrm{boundary}}(v) \tag{2.9}$$

the algorithm triggers a refinement. This simple rule refines each geometric element up to a given level ℓ, i.e. the formula yields an adaptive Cartesian grid with approximation order $\mathcal{O}(h)$ (Figure 2.10). The approximation of the adaptive Cartesian grid

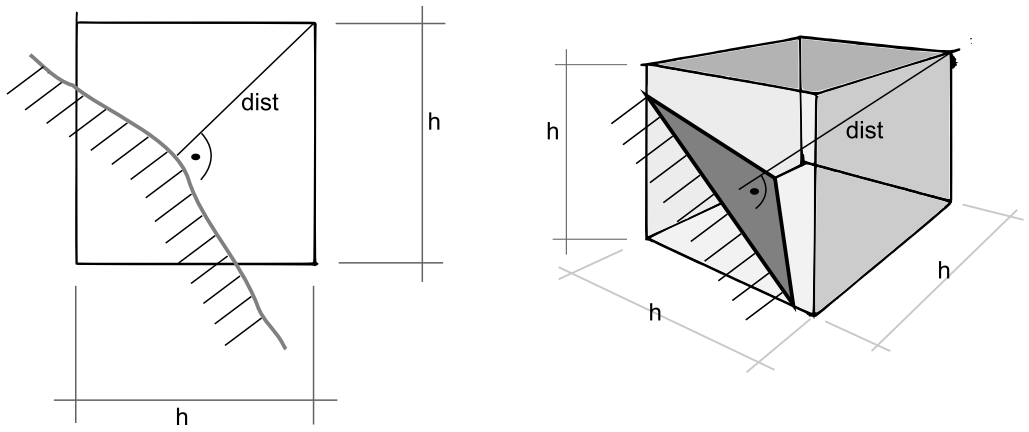


Figure 2.10: The computational domain's boundary intersects a square ($d = 2$, left) or cube ($d = 3$, right). The maximum distance from a hypercube's vertex to the boundary is $dist \leq \sqrt{d \cdot h^2}$. The spatial discretisation error is in $\mathcal{O}(h)$.

resulting from the refinement criterion above thus is in $\mathcal{O}\left(k^{-\ell}\right)$, if the computational domain is embedded into the unit hypercube.

Although the k-spacetree approximates the computational domain up to any precision, the computational domain's boundary does not match the continuous boundary exactly, and a solver has to project the continuous boundary's conditions to the computational grid's boundary vertices. In this thesis, I apply a simple projection: For each boundary point, I search for the nearest point on $\partial\Omega$ and copy its state and value, i.e. I map information along the distance vector to the boundary (Figure 2.10). A more sophisticated projection preserving the overall boundary's L_2-norm, e.g., might be of great value for some applications.

The two-step refinement cascade in Section 2.6 makes the k-spacetree's structure invariant throughout the traversal (Figure 2.8). `refinement-triggered`, an intermediate state, encapsulates the refinement mechanism and constraints, and it enables any algorithm to trigger the refinement anytime. As a result, at most one grid level is added per traversal. Another approach preserving a constant number of vertex accesses for $v \notin \mathbb{H}_{\mathcal{T}}$ would be to allow a refinement only immediately before the first time a vertex is read in a traversal. Criterion (2.9) depends exclusively on the vertex's position, i.e. the spacetree construction can evaluate it whenever a vertex is created, and it can switch the vertex's state immediately to refined. It would fit to a modified spacetree construction without the intermediate state.

Both approaches come along with pros and cons: If an algorithm requires a spacetree with height ℓ and does not modify the grid later on throughout the traversal, it does not make sense to spend ℓ traversals to build up the grid. In turn, a multigrid

F-cycle [73] for a sufficient smooth problem[9] never requires more than one additional level per traversal. Peano follows solely a one-level-per-traversal policy.

A PDE defines the type of the boundary's vertices and fitting boundary values. Thus, the numerical scheme is able to reconstruct the vertices' state in each iteration on-the-fly, and there is no need to hold boundary vertices persistently. The codes coming along with forerunners of this thesis, ([35, 39] and [63]), hence never store boundary vertices. Experience with longer simulation runs and increasingly complex applications reveals that the vertex construction process consumes more runtime than the overhead resulting from boundary vertices stored explicitly. Furthermore, the multiscale solvers in this thesis modify the boundary's vertices on coarser levels frequently. An on-the-fly reconstruction of this modification is cumbersome, as it demands for the complete boundary's multilevel data. I thus decided to store boundary vertices, too.

Examining the computational domain $\Omega = (0,1)^d$ delivers another insight: Here, the spacetree's root element e_0 equals the computational domain ($e_0 = \Omega$), and all the spacetree's boundary vertices are hanging vertices. According to Definition 2.3, hanging vertices hold no semantics, i.e. the whole boundary condition is to be represented by e_0's vertices. This is neither desirable nor always possible. As soon as the boundary vertices' state or value is not invariant in space, the hanging vertices should hold information on their own. I thus refine e_0 once or twice until I am able to embed Ω into an element surrounded by other (empty) elements. For $k \geq 3$, one refinement step is sufficient. This ensures that no boundary vertex is hanging.

2.8 Traversal Events

k-spacetrees and the element-wise k-spacetree traversals build the basis of the subsequent three chapters. The algorithms plug into the transitions of the traversal as well as into the steps of the grid generation. Herefrom, they evaluate the formulas to compute the PDE's solution, visualise data, postprocess results, update the spacetree itself, setup initial solution guesses as well as boundary conditions, and so forth. The subsequent section formalises the plug-in mechanism.

In the implementation, the plug-in mechanism equals a template method pattern [27], i.e. the software defines a number of operations with a predefined signature. The traversal then calls these operations at the right time passing the required arguments. I call these operations *events*.

The events are split up into two groups. Events of the first group correspond to the geometry management and the grid generation process. They are called throughout the grid construction and enlisted in Table 2.1. Yet, as any algorithm is allowed to update the grid structure anytime, the grid construction overlaps with other phases.

[9] For singularities, this statement does not hold anymore.

Events of the second group correspond to the spacetree traversal. They map the transitions within the hierarchical grid to operations typically belonging to the solver or data postprocessing.

Table 2.1: Grid generation events.

Event	Description
$createDegreeOfFreedom$ $\mathbb{R}^d \times \mathbb{R} \times \mathbb{V}_{\mathcal{T}} \mapsto \mathbb{V}_{\mathcal{T}}$ $(x, h, v) \mapsto v'$	*Factory method.* The method is given a new vertex (hull) v. The user's implementation initialises this vertex and returns v'. Besides the vertex record, the operation accepts the position x and mesh width h corresponding to the vertex's level.
$createDegreeOfFreedom$ $\mathbb{R}^d \times \mathbb{R} \times \mathbb{V}_{\mathcal{T}} \mapsto \mathbb{E}_{\mathcal{T}}$ $(x, h, e) \mapsto e'$	*Factory method.* The method is given a new geometric element (hull) e. The user's implementation initialises this element and returns e'. x's and h's semantics equals the other $createDegreeOfFreedom$ operation.
$isElementOutsideDomain$ $\mathbb{R}^d \times \mathbb{R}^d \mapsto \{\top, \bot\}$ $(x, h) \mapsto b$	*Spatial query.* The method is given a point x in space and a (hyper-)rectangularly h-environment. It returns whether or not the point identified by the position and the surrounding environment is outside the computational domain. The grid uses this information for optimisation purposes according to Corollary 2.1.
$isElementInsideDomain$ $\mathbb{R}^d \times \mathbb{R}^d \mapsto \{\top, \bot\}$ $(x, h) \mapsto b$	*Spatial query.* Counterpart to the event before. Only geometric elements completely inside the computational domain belong to the computational grid.
$refine$ $\mathbb{R}^d \times \mathbb{R}^d \mapsto \{\top, \bot\}$ $(x, h) \mapsto b$	*Spatial query.* The method is given a point x in space and a (hyper-)rectangularly h-environment. It returns whether the grid should refine here. The geometric refinement criterion e.g. plugs into this event.

The transition events (Table 2.2) are split up into events corresponding to the overall traversal, events corresponding to one single level, events representing an inter-level transition, and operations managing the lifecycle of geometric elements and vertices. All the events' implementations are allowed to modify the elements' and vertices' state. They are not allowed to modify spatial data such as positions, levels and mesh widths.

Before and after a traversal, the grid triggers the events $beginTraversal$ and $endTraversal$. For one single level, $enterElement$ accepts an element, 2^d vertices, the position of vertex v_0[10], the element's size and level. It is called before the traversal enters the element. $leaveElement$ is the counterpart. $touchVertexFirstTime$ is invoked the first time a vertex v is read throughout the traversal. Besides the position, the level and the corresponding mesh width h, the traversal also passes the parent vertices and an integer vector p identifying the vertex v's discrete position within a $(k+1)^d$ Cartesian grid corresponding to the geometric parent element. As all the traversals in this thesis are based upon a depth-first traversal preserving the child-father relationship, the value p is always available to the traversal implementation. $touchVertexLastTime$ is the counterpart of $touchVertexFirstTime$. If it is triggered, all of v's adjacent geometric elements have been traversed before.

The set of inter-level transition events comprises two operations: $loadSubElement$ precedes a top-down transition. $storeSubElement$ corresponds to a bottom-up transition. Both operations accept a refined geometric element, the element's spatial attributes (position, level, etc.) and its vertices. Furthermore, they get all the data of a child element that is loaded or stored, respectively. The operations facilitate an inter-level information transfer.

The operations $createPersistentVertex$ mirrors the $createDegreeOfFreedom$ factory method for vertices. It is a redundant lifecycle management operation that enables the programmer to implement a whole lifecycle algorithm within the grid traversal event set. $destroyPersistentVertex$ is the counterpart of $createPersistentVertex$. Both operations are defined on $\mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$. The operations $createTemporaryVertex$ and $destroyTemporaryVertex$ in turn provide the same plug-in possibility for hanging vertices $v \in \mathbb{H}_{\mathcal{T}}$.

The traversal events permit the interpretation of a traversal as event sequence. The event sequence corresponding to a depth-first traversal is given by Algorithm 2.4. For each algorithm implemented as plain set of operations, its behaviour is well-defined as mapping from events to this operations. To give such a mapping for each algorithm thus proves that the algorithm fits into the k-spacetree traversal concept. The element-wise traversal poses restrictions on the information available throughout the traversal. This information restriction is also formalised by the event sequence, and a mapping proves that an algorithm's implementation can make ends meet with the information available. Besides the data passed to the events, each mapping also has to consider the order of the events within the k-spacetree traversal. The order's properties can be formalised as invariants. For their obviousness, I decided not to write them down explicitly but to give some simple examples.

[10] See the excursus on page 29 for a description of the enumeration and the position indexing.

Table 2.2: Grid traversal events.

Operation Name	**Arguments**
$beginTraversal$	$\{\top, \bot\} \mapsto \emptyset$
$endTraversal$	$\emptyset \mapsto \emptyset$
$enterElement$	$\mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \mapsto \mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}}$ $(v, e, h, x, l) \mapsto (v', e')$
$leaveElement$	$\mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \mapsto \mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}}$ $(v, e, h, x, l) \mapsto (v', e')$
$touchVertexFirstTime$	$\mathbb{V}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \{0, k\}^d \times \mathbb{V}_{\mathcal{T}}^{2^d}$ $(v, x, h, l, p, v_{\text{parent}}) \mapsto \left(v', v'_{\text{parent}}\right)$
$touchVertexLastTime$	$\mathbb{V}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \{0, k\}^d \times \mathbb{V}_{\mathcal{T}}^{2^d}$ $(v, x, h, l, p, v_{\text{parent}}) \mapsto \left(v', v'_{\text{parent}}\right)$
$loadSubElement$	$\mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{R}^d \mapsto$ $\mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}}$ $(v_{\text{parent}}, e_{\text{parent}}, h_{\text{parent}}, x_{\text{parent}}, l_{\text{parent}}, v_{\text{child}}, e_{\text{child}}, x_{\text{child}}) \mapsto$ $\left(v'_{\text{parent}}, e'_{\text{parent}}, v'_{\text{child}}, e'_{\text{child}}\right)$
$storeSubElement$	$\mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{R}^d \mapsto$ $\mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}} \times \mathbb{V}_{\mathcal{T}}^{2^d} \times \mathbb{E}_{\mathcal{T}}$ $(v_{\text{parent}}, e_{\text{parent}}, h_{\text{parent}}, x_{\text{parent}}, l_{\text{parent}}, v_{\text{child}}, e_{\text{child}}, x_{\text{child}}) \mapsto$ $\left(v'_{\text{parent}}, e'_{\text{parent}}, v'_{\text{child}}, e'_{\text{child}}\right)$
$createPersistentVertex$	$\mathbb{V}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \{0, k\}^d \times \mathbb{V}_{\mathcal{T}}^{2^d} \mapsto \mathbb{V}_{\mathcal{T}}$ $(v, x, h, l, p, v_{\text{parent}}) \mapsto v'$
$destroyPersistentVertex$	$\mathbb{V}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \{0, k\}^d \times \mathbb{V}_{\mathcal{T}}^{2^d} \mapsto \mathbb{V}_{\mathcal{T}}$ $(v, x, h, l, p, v_{\text{parent}}) \mapsto v'$
$createTemporaryVertex$	$\mathbb{V}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \{0, k\}^d \times \mathbb{V}_{\mathcal{T}}^{2^d} \mapsto \mathbb{V}_{\mathcal{T}}$ $(v, x, h, l, p, v_{\text{parent}}) \mapsto v'$
$destroyTemporaryVertex$	$\mathbb{V}_{\mathcal{T}} \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{N}_0 \times \{0, k\}^d \times \mathbb{V}_{\mathcal{T}}^{2^d} \mapsto \mathbb{V}_{\mathcal{T}}$ $(v, x, h, l, p, v_{\text{parent}}) \mapsto v'$

Algorithm 2.4 The depth-first traversal triggers a well-defined set of events. Algorithm 2.1 defines the depth-first traversal using a stack. Here, I prefer a recursive formulation. Both are equivalent as the stack in Algorithm 2.1 mirrors a call stack. As I omit the event's arguments and technical details, $touchVertexFirstTime$ and $touchVertexLastTime$ as well as the lifecycle and geometry management events do not occur in the description. e_0 is the root node of the k-spacetree .

```
1: procedure traverseDfs
2:     trigger beginTraversal
3:     dfs(e_0)
4:     trigger endTraversal
5: end procedure
6: procedure dfs(e)
7:     trigger enterElement
8:     for e_i ⊑_child e do
9:         trigger loadSubElement
10:        dfs(e_i)
11:        trigger storeSubElement
12:    end for
13:    trigger leaveElement
14: end procedure
```

- The event $touchVertexLastTime$ ensures that $leaveElement$ has been called for each adjacent geometric element.
- $enterElement$ is called before $leaveElement$.
- Both $enterElement$ and $leaveElement$ are triggered exactly once per traversal.
- The event $enterElement$ ensures that for each vertex either $touchVertexFirstTime$, $createPersistentVertex$ or $createTemporaryVertex$ has been called before.

A light-weight component composition following the publish-subscribe idea of the observer pattern [27] puts the event concept into the code: The grid management and traversal component defines interfaces comprising the traversal and grid generation events. These interfaces are the only interaction points—a separated interface [24]—visible from outside the component. Plug-ins implement them in their own components and delegate the events to operations of their own.

The composition pattern on the one hand enables the algorithm to exchange the mapping of events to an algorithm's operations at runtime. A programmer is thus able to exchange the algorithm throughout the computation (perform γ_1 solver

traversals, plug in a plotter afterwards and stream the results to a visualisation component, and perform γ_2 data postprocessing traversals afterwards, e.g.). Furthermore, the mapping from events to operations has arbitrary cardinality, since there is the opportunity to make one event trigger several algorithms' operations: An additional event interface implementation just has to follow the multiple dispatch paradigm [27] and delegate one event to several implementations. In this thesis, e.g., the parallelisation and the Poisson solver algorithms are completely independent of each other. Yet, it is straightforward to combine them: I just plug-in both of them into the event sequence, i.e. each event triggers the solver's operations as well as the parallel algorithm's operations.

Albeit Peano's mapping from events to operations is flexible and follows the separation-of-concerns paradigm, it does not lead to a performance breakdown if it is realised by static polymorphism—a pattern often found in object-oriented high-performance code ([4], e.g.). Here, lookup table techniques typically employed for virtual function tables are replaced by static binding due to generic programming. To have all possible combinations of plugins available at compile time is the only consequence arising from the static polymorphism. In my experiments, this was always the case. Nevertheless, Peano also offers a plug-in mechanism for dynamic polymorphism.

2.9 Experiments

The following experiments analyse different spacetree properties for three out of four different geometries from Figure 2.2. These experiments either fit exactly to the spacetree's structure (the cube equals the spacetree's root element), exhibit a complex surface not fitting to the spacetree elements (sphere), or hold faces not aligned with the spacetree's faces (L-shape).

All measurements are based upon $(k = 3)$-partitioning. The L-shape's continuous domain is $(0,1)^d - \left(0, \frac{1}{2}\right)^d$, and the spacetree's hypercubes near $x_i = \frac{1}{2}$ consequently never fit exactly to the computational domain. As a result, the spacetree has to approximate the computational domain although all faces are aligned with the Cartesian coordinate system's axes. The sphere is embedded into the unit hypercube, too, and its analytical volume thus equals $\frac{1}{2^d} \cdot \frac{\pi^{d/2}}{\Gamma(d/2+1)}$ with the gamma function Γ.

First, the volume of the continuous computational domain is contrasted with the volume of the fine grid of the k-spacetree . The comparison is trivial for the hypercube as computational domain, since a hypercube equals the spacetree's root element. Both domains' volumes equal for spacetrees of arbitrary height, and, thus, a plot comparing both volumes would be without information (see Tables 2.3 and 2.5 instead). If the L-shape with $\Omega = (0,1)^d \setminus (0, \frac{1}{2})^d$ is embedded into the $(k = 3)$-spacetree 's root element,

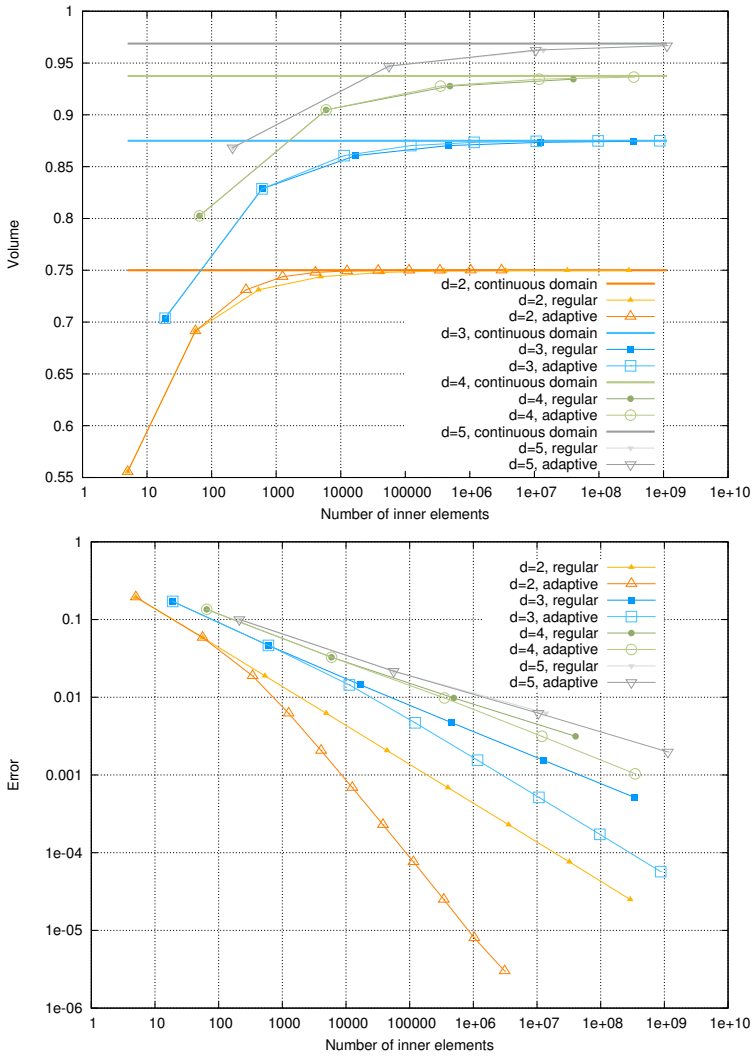


Figure 2.11: Volume of spatial discretisation of the L-shape from Figure 2.2. Each subsequent tick corresponds to one refinement step.

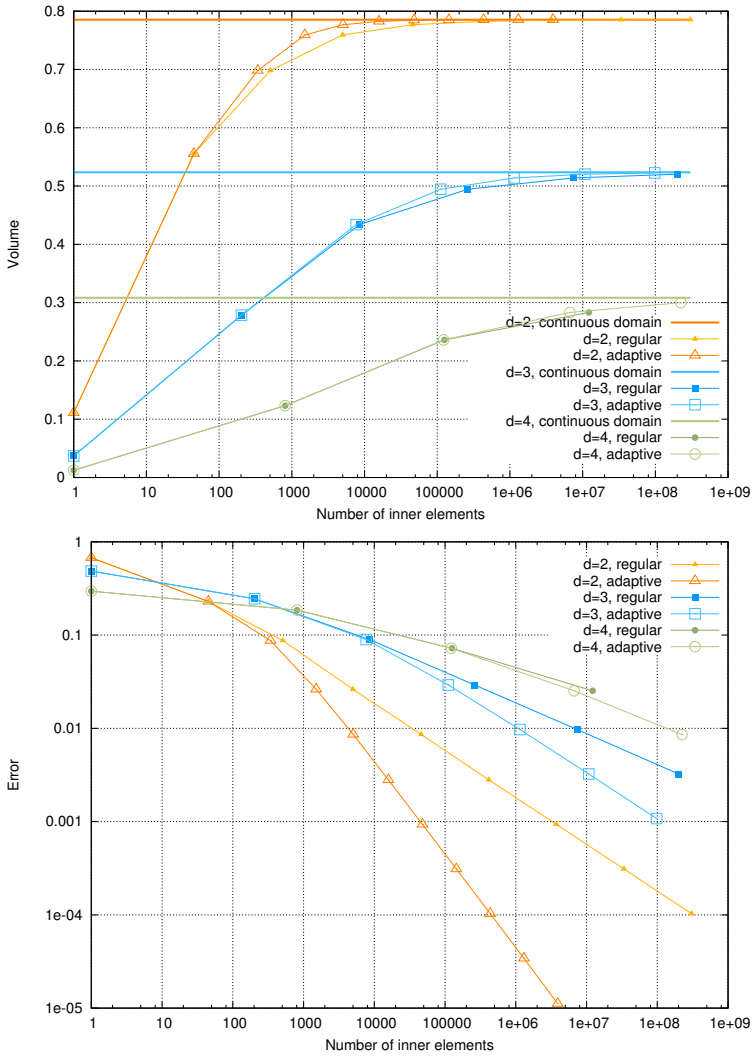


Figure 2.12: Experiment from Figure 2.11 with a hypersphere as computational domain. Each subsequent tick corresponds to one refinement step.

Table 2.3: Hypercube domain with cardinalities of elements and vertices, as well as resulting memory overhead. For each dimension, the upper block gives figures for a regular grid, the lower block for an adaptive grid based upon the geometric refinement criterion.

	fine grid Ω_h			whole k-spacetree		
d	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{E}_\mathcal{T})\rvert$	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{V}_\mathcal{T})\rvert$	boundary vertices	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{E}_\mathcal{T})\rvert$	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{V}_\mathcal{T})\rvert$	boundary vertices
2	43,046,721	47,812,196	34,992	48,427,561	48,407,888	39,364
	387,420,489	430,414,724	104,976	435,848,050	435,789,012	118,096
	944,457	708,340	314,928	1,062,514	708,340	354,292
	2,833,993	2,125,492	944,784	3,188,242	2,125,492	1,062,880
3	14,348,907	14,684,488	393,664	14,900,788	14,702,584	398,592
	387,420,489	400,000,840	3,542,944	402,321,277	400,530,936	3,587,240
	92,928,291	64,334,840	31,886,464	96,502,456	64,334,840	32,285,056
	838,389,319	580,423,224	286,978,144	870,635,062	580,423,224	290,565,384
4	531,441	461,072	163,584	538,084	461,088	163,840
	43,046,721	41,416,976	4,409,856	43,584,805	41,421,088	4,416,016
	11,604,561	7,824,400	4,409,856	11,749,618	7,824,400	4,416,016
	339,335,761	229,023,760	119,045,376	343,577,458	229,023,760	119,209,216
5	9,049	32,800	68,224	59,293	32,800	68,256
	14,348,907	11,914,144	5,396,224	14,408,200	11,914,176	5,397,248
	10,281,371	6,759,520	5,396,224	10,323,856	6,759,520	5,397,248
	1,119,435,615	748,034,144	435,927,424	1,124,061,382	748,034,144	435,995,680

- the volume of the fine grid converges to the analytical volume linearly in h: Each tick in the figures represents one experiment. The heights of two subsequent experiments' spacetrees differ by one, i.e. each refinement reduces the error by a factor of $k = 3$ (Figure 2.11, top). Two subsequent refinements thus reduce the volume's error almost by one decade (Figure 2.11, bottom), and the convergence rate is independent of both h and d.

- As the fine grid is contained within the computational domain, the fine grid's volume is smaller than the analytical volume. Hence, the fine grid's volume monotonically increases to the actual solution.

- The convergence rate depends on d, as there's a constant $\sqrt{d}$ hidden in the error estimate (Figure 2.10): With increasing dimension d, the convergence speed deteriorates.

- The grids for adaptive spacetrees yield the same spatial discretisation error as their regular counterpart. Two ticks for the same d with same ordinate value represent such a pair of experiments. The adaptive grid comes along with a substantially smaller number of elements and vertices.

If a hypersphere is embedded into the k-spacetree's root element, the same observations hold (Figure 2.12).

Second, the cardinalities of the spacetree's inner fine grid vertices and elements are compared with the overall number of inner elements and vertices in the whole k-spacetree. As there is no spatial discretisation error for the cube, the cube experiment reveals the pristine overhead resulting from the additional levels (Table 2.3), which is, in fact, not an overhead but the additional memory needed to store the coarser grid levels in the spacetree.

With the k-spacetree holding all the levels simultaneously, the overall number of vertices and elements outnumbers the fine grid numbers by a factor of at most $\frac{k^d}{k^d-1}$: For a given fine grid level ℓ, the number of elements and vertices on level $\ell-1$ is smaller by a factor of k^d. The overall number of elements and vertices thus is bounded by a factor of

$$\begin{aligned} 1 + k^d + k^{2d} + k^{3d} + \ldots &= \sum_{i=0}^{\ell} k^{i \cdot d} \\ &= k^{\ell \cdot d} \cdot \sum_{i=0}^{\ell} k^{(i-\ell) \cdot d} = k^{\ell \cdot d} \cdot \sum_{i=0}^{\ell} \left(\frac{1}{k^d} \right)^i \\ &\leq k^{\ell \cdot d} \cdot \sum_{i=0}^{\infty} \left(\frac{1}{k^d} \right)^i = k^{\ell \cdot d} \cdot \frac{1}{1 - 1/k^d}. \end{aligned}$$

The boundary of the computational domain is a $d-1$-manifold. Thus, the overhead of additional boundary vertices on the coarser grids is smaller than the overhead for the inner vertices and elements. The tables for the L-shape and the sphere reveal the same conclusions for more complicated domains (Table 2.4).

Third, the total number of the spacetree's inner elements is contrasted with the total number of spacetree elements. For persistent boundary vertices, the algorithm refines the $(k = 3)$-spacetree once and embeds the computational domain into the central element. This leads to an $k^3 - 1$ additional geometric elements on the first refinement level—some kind of shadow boundary layer. All elements of the shadow layer are outside the computational domain. They simplify and speed up the handling of the boundary vertices but bring along a memory overhead. As only vertices inside the computational domain and lying on the discretised domain's boundary carry a refinement flag, the shadow layer elements are refined if and only if they are adjacent to a boundary vertex. The shadow layer is adaptive. Furthermore,

it corresponds to the domain's surface—a submanifold. The overhead is therefore bounded by a constant smaller than $\frac{k}{k-1}$ (see Table 2.5 for the hypercube, Table 2.6 for the L-shape domain).

The cube fits exactly to the spacetree's root element. All the vertices along the root element's faces are boundary vertices and, hence, refined. The maximum refinement level of the grid follows the cube's faces exactly, and the refinement structure left and right, above or below, and so forth of the boundary is exactly the same. It gives the worst-case overhead resulting from the embedding. The sphere's overhead (Table 2.6) is smaller than the cube's overhead, as the sphere's surface compared to its volume is smaller than the cube's surface compared to its volume.

Table 2.4: Experiment from Table 2.3 with an L-shape as computational domain (upper part) or a hypersphere as computational domain (lower part).

	fine grid Ω_h			whole k-spacetree		
d	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{E}_\mathcal{T})\rvert$	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{V}_\mathcal{T})\rvert$	boundary vertices	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{E}_\mathcal{T})\rvert$	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{V}_\mathcal{T})\rvert$	boundary vertices
2	32,281,760	35,850,399	34,992	36,315,748	36,296,076	39,363
	290,555,525	322,784,799	104,976	326,871,273	326,812,236	118,095
	1,033,016	796,903	314,928	1,151,069	796,903	354,291
	3,099,701	2,391,204	944,784	3,453,946	2,391,204	1,062,879
3	12,533,059	12,799,719	393,664	13,013,141	12,814,938	398,591
	338,793,364	349,557,867	3,542,944	351,806,505	350,016,165	3,587,239
	96,973,571	68,380,150	31,886,464	100,547,728	68,380,150	32,285,055
	874,739,326	616,773,261	286,978,144	906,985,061	616,773,261	290,565,383
4	493,025	422,031	163,584	499,026	422,031	163,839
	40,220,960	38,552,799	4,409,856	40,719,986	38,556,270	4,416,015
	11,902,160	8,122,159	4,409,856	12,047,201	8,122,159	4,416,015
	346,986,545	236,674,704	119,045,376	351,228,226	236,674,704	119,209,215
5	55,924	29,643	68,224	56,135	29,643	68,255
	13,811,083	11,373,195	5,396,224	13,867,218	11,373,195	5,397,247
	55,924	29,643	68,224	56,135	29,643	68,255
	10,500,039	6,978,938	5,396,224	10,542,492	6,978,938	5,397,247
2	33,795,465	37,530,044	34,976	38,014,840	37,995,200	39,296
	304,239,481	337,982,468	104,960	342,254,321	342,195,320	118,020
	1,304,557	1,068,512	314,912	1,422,542	1,068,512	354,208
	3,907,249	3,198,828	944,768	4,261,418	3,198,828	1,062,788
3	7,373,533	7,484,720	305,104	7,645,057	7,491,896	308,440
	201,599,327	207,590,440	2,770,648	209,244,384	207,845,688	2,803,872
	10,734,471	8,287,640	2,770,648	11,038,572	8,287,640	2,803,872
	97,961,183	75,594,064	25,007,824	100,751,974	75,594,064	25,316,264
4	125,481	94,512	70,208	126,291	94,512	70,224
	12,187,065	11,259,792	2,171,392	12,313,356	11,260,032	2,173,088
	6,675,865	5,036,464	2,171,392	6,733,266	5,036,464	2,173,088
	223,727,577	169,392,176	61,093,056	225,768,354	169,392,176	61,163,280

Table 2.5: Memory overhead of hypercube due to embedding: Total number of inner elements/vertices compared to total number of elements/vertices in the spacetree. For each dimension, the upper block gives figures for a regular grid, the lower block for an adaptive grid due to the geometric refinement criterion.

d	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{E}_{\mathcal{T}})\rvert$	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{V}_{\mathcal{T}})\rvert$	$\lvert\mathbb{E}_{\mathcal{T}}\rvert$	$\lvert\mathbb{V}_{\mathcal{T}}\rvert$
2	$4.84 \cdot 10^7$	$4.84 \cdot 10^7$	$4.85 \cdot 10^7$	$4.85 \cdot 10^7$
	$4.36 \cdot 10^8$	$4.36 \cdot 10^8$	$4.36 \cdot 10^8$	$4.36 \cdot 10^8$
	$1.06 \cdot 10^6$	$7.08 \cdot 10^5$	$1.77 \cdot 10^6$	$2.13 \cdot 10^6$
	$3.19 \cdot 10^6$	$2.13 \cdot 10^6$	$5.31 \cdot 10^6$	$6.38 \cdot 10^6$
3	$1.49 \cdot 10^7$	$1.47 \cdot 10^7$	$1.59 \cdot 10^7$	$1.61 \cdot 10^7$
	$4.02 \cdot 10^8$	$4.01 \cdot 10^8$	$4.11 \cdot 10^8$	$4.13 \cdot 10^8$
	$9.65 \cdot 10^7$	$6.44 \cdot 10^7$	$1.61 \cdot 10^8$	$1.94 \cdot 10^8$
	$8.71 \cdot 10^8$	$5.80 \cdot 10^8$	$1.45 \cdot 10^9$	$1.74 \cdot 10^9$
4	$5.38 \cdot 10^5$	$4.61 \cdot 10^5$	$1.09 \cdot 10^6$	$1.24 \cdot 10^6$
	$4.36 \cdot 10^7$	$4.14 \cdot 10^7$	$5.58 \cdot 10^7$	$5.85 \cdot 10^7$
	$1.18 \cdot 10^7$	$7.82 \cdot 10^7$	$2.22 \cdot 10^7$	$2.67 \cdot 10^7$
	$3.44 \cdot 10^8$	$2.29 \cdot 10^8$	$5.96 \cdot 10^8$	$7.16 \cdot 10^8$
5	$5.92 \cdot 10^4$	$3.28 \cdot 10^4$	$5.70 \cdot 10^5$	$8.19 \cdot 10^5$
	$1.44 \cdot 10^7$	$1.19 \cdot 10^7$	$3.41 \cdot 10^7$	$4.00 \cdot 10^7$
	$5.91 \cdot 10^4$	$3.18 \cdot 10^4$	$5.70 \cdot 10^5$	$8.18 \cdot 10^5$
	$1.03 \cdot 10^7$	$6.76 \cdot 10^6$	$2.90 \cdot 10^7$	$3.59 \cdot 10^7$

Table 2.6: Experiment from Table 2.5 with an L-shape as computational domain (upper part) and a hypersphere as computational domain (lower part).

d	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{E}_{\mathcal{T}})\rvert$	$\lvert\mathcal{P}_{\text{inside}}(\mathbb{V}_{\mathcal{T}})\rvert$	$\lvert\mathbb{E}_{\mathcal{T}}\rvert$	$\lvert\mathbb{V}_{\mathcal{T}}\rvert$
2	$3.63 \cdot 10^7$	$3.63 \cdot 10^7$	$3.64 \cdot 10^7$	$3.64 \cdot 10^7$
	$3.27 \cdot 10^8$	$3.27 \cdot 10^8$	$3.27 \cdot 10^8$	$3.27 \cdot 10^8$
	$1.15 \cdot 10^6$	$7.97 \cdot 10^5$	$1.77 \cdot 10^6$	$2.13 \cdot 10^6$
	$3.45 \cdot 10^6$	$2.39 \cdot 10^6$	$5.31 \cdot 10^6$	$6.38 \cdot 10^6$
3	$1.30 \cdot 10^7$	$1.28 \cdot 10^7$	$1.40 \cdot 10^7$	$1.42 \cdot 10^7$
	$3.52 \cdot 10^8$	$3.50 \cdot 10^8$	$3.60 \cdot 10^8$	$3.62 \cdot 10^8$
	$1.01 \cdot 10^8$	$6.84 \cdot 10^7$	$1.61 \cdot 10^8$	$1.94 \cdot 10^8$
	$9.07 \cdot 10^9$	$6.17 \cdot 10^8$	$1.45 \cdot 10^9$	$1.74 \cdot 10^9$
4	$5.00 \cdot 10^5$	$4.22 \cdot 10^5$	$1.04 \cdot 10^6$	$1.19 \cdot 10^6$
	$4.07 \cdot 10^7$	$3.86 \cdot 10^7$	$5.26 \cdot 10^7$	$5.54 \cdot 10^7$
	$1.21 \cdot 10^7$	$8.12 \cdot 10^6$	$2.22 \cdot 10^7$	$2.67 \cdot 10^7$
	$3.51 \cdot 10^8$	$2.37 \cdot 10^8$	$5.96 \cdot 10^8$	$7.16 \cdot 10^8$
5	$5.61 \cdot 10^4$	$2.96 \cdot 10^5$	$5.63 \cdot 10^5$	$8.11 \cdot 10^5$
	$1.39 \cdot 10^7$	$1.14 \cdot 10^7$	$3.34 \cdot 10^7$	$3.92 \cdot 10^7$
	$5.61 \cdot 10^4$	$2.96 \cdot 10^4$	$5.63 \cdot 10^5$	$8.11 \cdot 10^5$
	$1.05 \cdot 10^7$	$6.98 \cdot 10^6$	$2.90 \cdot 10^7$	$3.59 \cdot 10^7$
2	$3.80 \cdot 10^7$	$3.80 \cdot 10^7$	$3.81 \cdot 10^7$	$3.81 \cdot 10^7$
	$3.42 \cdot 10^8$	$3.42 \cdot 10^8$	$3.42 \cdot 10^8$	$3.43 \cdot 10^8$
	$1.42 \cdot 10^6$	$1.07 \cdot 10^6$	$1.77 \cdot 10^6$	$2.13 \cdot 10^6$
	$4.26 \cdot 10^6$	$3.20 \cdot 10^6$	$5.31 \cdot 10^6$	$6.38 \cdot 10^6$
3	$7.65 \cdot 10^6$	$7.49 \cdot 10^6$	$8.11 \cdot 10^6$	$8.27 \cdot 10^6$
	$2.09 \cdot 10^8$	$2.08 \cdot 10^8$	$2.13 \cdot 10^8$	$2.15 \cdot 10^8$
	$1.10 \cdot 10^7$	$8.29 \cdot 10^6$	$1.39 \cdot 10^7$	$1.67 \cdot 10^7$
	$1.01 \cdot 10^8$	$7.56 \cdot 10^7$	$1.26 \cdot 10^8$	$1.51 \cdot 10^8$
4	$1.26 \cdot 10^5$	$9.45 \cdot 10^4$	$2.55 \cdot 10^5$	$3.17 \cdot 10^5$
	$1.23 \cdot 10^7$	$1.13 \cdot 10^7$	$1.58 \cdot 10^7$	$1.71 \cdot 10^7$
	$6.73 \cdot 10^6$	$5.04 \cdot 10^6$	$9.53 \cdot 10^6$	$1.15 \cdot 10^7$
	$2.26 \cdot 10^8$	$1.69 \cdot 10^8$	$2.94 \cdot 10^8$	$3.52 \cdot 10^8$
5	$2.94 \cdot 10^3$	$\perp$	$6.56 \cdot 10^4$	$1.18 \cdot 10^5$
	$1.64 \cdot 10^6$	$\perp$	$4.36 \cdot 10^6$	$5.79 \cdot 10^6$
	$2.94 \cdot 10^3$	$\perp$	$6.56 \cdot 10^4$	$1.18 \cdot 10^5$
	$1.63 \cdot 10^6$	$\perp$	$4.33 \cdot 10^6$	$5.78 \cdot 10^6$

2.10 Outlook

This chapter defines the grid underlying the whole thesis. Its definition is complete and closed with respect to the algorithms here. Nevertheless, many straightforward extensions and generalisations exist. This closing section enlists some of them. The list is neither complete nor representative.

One extension applying the spacetree philosophy further is the *boundary extended spacetree* [25] providing an improved boundary approximation. While [25] constructs them with $(k = 2)$-spacetrees in the three-dimensional case, the extension to arbitrary k is straightforward. The spacetree cube's faces are discretised recursively in d by a $d-1$-dimensional spacetree. Applying the idea for boundary faces resolves the boundary with a higher order and enables the application to reduce the discretisation error. An improved boundary resolution is of great value in many PDE problems, if the computational domain exhibits a complicated shape or if the boundary values' precision is of great importance. Naively refining the spacetree's elements at the boundary in fact does not work properly for $d \geq 3$: Let a numerical scheme converge in $\mathcal{O}(h^2)$. As the boundary's approximation order is in $\mathcal{O}(h)$, halving the mesh within the domain entails the boundary's elements to half twice, i.e. $h \mapsto \frac{h}{2} \mapsto \frac{h}{4}$. Otherwise, the boundary's approximation error pollutes the overall solution. Such a refinement increases the number of geometric elements within the computational domain by a factor of k^d. As the boundary's "dimension" equals $d-1$, it increases the number of boundary elements by a factor of $k^{2(d-1)}$ due to the two refinement steps. The boundary elements thus soon dominate the overall number of geometric elements, the memory consumption, and the computational load if the domain shape is sufficiently complicated.

Besides an improvement of the boundary accuracy, boundary extended spacetrees also fit to hypercube faces lying inside the computational domain. Applying them there mirrors construction ideas of sparse grids [15] and yields promising results for example for multigrid solvers on convection dominated problems [1].

Accuracy at the boundary is important for the k-spacetree's coarser levels, too. A loss of precision on coarser levels does not affect the solution's accuracy, but influences the solver of the linear equation system: The coarser the grid, the smaller the computational domain becomes. Thus, geometric multigrid algorithms can not correct the solution at the boundary as efficiently as they do within the computational domain.

Another boundary improvement adds information where the edges intersect the computational domain to each boundary vertex. Such a scheme also reduces the discretisation error and improves the multigrid convergence rate. [75], e.g., discusses this approach for a $(k = 3)$-spacetree environment and three-dimensional problems. The extension to arbitrary k and dimension d is straightforward.

The k-spacetree definition in this thesis does not restrict the level difference for

hanging vertices: For a two-dimensional challenge, up to $k^{\Delta l} - 1$ hanging nodes can by placed on one edge. Δl is the level difference of two adjacent geometric elements on the fine grid. To extend this formula to arbitrary dimensions is trivial. Some numerical problems benefit from balanced grids, i.e. two adjacent fine grid elements' levels differ at most by one. Such a balanced adaptive Cartesian grid corresponds to a sufficiently balanced tree, and this balancing can be ensured by posing the invariant

$$\forall e \in \mathbb{E}_{\mathcal{T}} : \qquad \exists v_1 \in vertex(e) : \mathcal{P}_{\text{refined}}(v_1) \Leftrightarrow \forall v_2 \in vertex(e) : v_2 \notin \mathbb{H}_{\mathcal{T}}$$

on the tree or the refinement criterion, respectively.

Many PDE's solutions exhibit an anisotropic behaviour [73]. The representation of anisotropic solutions benefits from anisotropic grids as such grids come along with less vertices and elements compared to the adaptive Cartesian grids here, where each element exhibits the same spatial resolution along each coordinate axis. Furthermore, multigrid algorithms for anisotropic problems demand for specialised inter-level transfer operators and tailored smoothers. The inter-level transfer operators also benefit from anisotropic grids. To extend the k-spacetree definition for anisotropic grids equals a modified refinement predicate

$$\mathcal{P}_{\text{refined}} : \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} \times \{1, \ldots, d\} \mapsto \{\top, \bot\}$$

controlling the refinement along each coordinate axis. The formalisms then become more complicated but all the principles remain the same.

The children of a refined k-spacetree element are both embedded into the father's hypercube and disjoint from each other. To remove the disjointness invariant is a more subtle change in the k-spacetree definition, but it permits data structures resolving complicated domains more accurately. Furthermore, it makes the grid fit to the grids used by many groups in the adaptive mesh refinement community [7, 62].

Throughout this dissertation, I refrain from all these generalisations. Instead, the pure k-spacetree introduced in this chapter is the basis of the subsequent three chapters. They present three different algorithms exploiting the k-spacetree 's definition: Chapter 3 introduces an efficient grid management, i.e. a management coming along with very low memory demands and good memory access behaviour. Chapter 5 establishes a parallelisation and load balancing approach that is able to cope with dynamic adaptive refinement. Chapter 4 implements a multiplicative multigrid with a full approximation storage scheme within the k-spacetree data structure. All three chapters are orthogonal, i.e. most of their presentation relies exclusively on this chapter and its definition. Besides Chapter 3, all algorithms holds for arbitrary k, and all three algorithmic ideas are well-suited for any dimension d.

3 Spacetree Traversal and Storage

Every PDE solver requires a traversal of the grid, and Chapter 2 shows that a traversal preserving the child-father relationship is of great value to process the k-spacetree. On the one hand, such a traversal facilitates a memory-modest encoding of the adaptive grid, as one refinement bit per vertex is sufficient. In each recursion step, the traversal determines due to this bit whether the recursion stops or continues, i.e. whether the geometric element is refined. On the other hand, the traversal's interplay of different recursion levels facilitates the implementation of inter-level transfer operations. They are essential for any multiscale algorithm.

Two traversals preserving the child-father relationship result from the depth-first and breadth-first order. Both come along with advantages and disadvantages. In this chapter, I concentrate on depth-first algorithms, as their simple backtracking mechanism affords plain and elegant recursive implementations, and their realisation itself comes along without any additional data container—the backtracking, i.e. the bottom-up traversal steps, is realised by the system's call stack. Despite this simplicity, the question nonetheless remains open what a well-suited data container for the grid's constituents, i.e. the vertices and geometric elements, as well as their connectivity information looks like. This chapter presents one solutions and discusses the storage of vertex and hypercube associated data. Besides the vertex refinement flag and the geometric encoding, this data comprises PDE-specific properties, too.

It turns out that the single bit for the spacetree code is also sufficient to store the complete structural information, i.e. the bidirectional vertex-element adjacency relations, if $k = 3$—a restriction finally weakened to odd k in the outlook. Although all the thesis' experiments are conducted for $k = 3$—I implemented only one code combining all the presented features—most insights hence hold for arbitrary odd k, and I thus a variable k whenever possible. The low memory requirements for the adjacency information and $k = 3$ is due to a sophisticated combination of the depth-first order with a space-filling curve and results in an exclusive usage of stacks as grid data container.

Without the careful selection of space-filling curves and their properties, a straightforward implementation of the data containers that does not restrict the adaptivity is a pointer-based data structure. Hereby, pointers hold the grid's adjacency and connectivity information. The most prominent example for such a data structure is the vef-graph in computer graphics [16]. Such pointer networks come along with at least three disadvantages:

1. The pointers need memory, i.e. the application's memory demands result from both the application's data plus the connectivity information. The pointers hence induce a memory overhead.

2. The size of pointer data structures is bounded by the overall memory address space available. To make an application work with pointer structures not fitting into this space anymore entails the development of a serialisation strategy. Besides the trickiness of such strategies, serialisation always comes along with some performance overhead. Due to that, problems not fitting into the main memory usually are considered as not solvable.

3. Pointer-based implementations rely on indirect memory access, i.e. the application does not read from the main memory directly. It first reads the record's address from the memory, and it second reads the actual record from this address. The two-step memory access causes a performance penalty. A more severe performance drawback results from the fact that the records typically are scattered among the address space. The mechanism thus exhibits non-local memory access behaviour and is not tuned to caches.

Numerous concepts realise the k-spacetree without pointers. Early approaches make both the refinement level and the spatial position act as access key (address) for records. The Morton ordering [57] for yields such a key. In this case, all the vertices are held in a global container and the address determines a position within this container. Hash tables are a natural choice for the container's realisation. A subtle choice of the key computation leads to a sophisticated and efficient memory access ([32], e.g.). Nevertheless, the approach still relies on indirect memory access and it is bounded by the available address space.

To overcome these constraints, the forerunners of this thesis [35, 39, 63] derive an alternative realisation concept. They fit the construction principle of a space-filling curve [66] into the depth-first traversal, and, thereof, they make the traversal use exclusively the two stack operations **push** and **pop** for both the vertex and the geometric element management. The number of stacks is fixed. Besides their simplicity, stacks exhibit three important memory access characteristics. The data access comes without indirect addressing, and the access itself offers both spatial and temporal locality: If a record is taken from or written to the memory, the subsequent memory access' position differs from the current position by at most one record's size. If a record is taken from or written to the memory, the time in-between two accesses is mall due to the small number of stacks and the curve's Hölder continuity. The latter two characteristics lead to a good cache hit rate [48].

Despite all these nice properties, the forerunners' algorithms suffer from a high implementation complexity: [35] derives an algorithm for a two-dimensional ($k = 3$)-spacetree using ten stacks. [63] extends this concept to $d = 3$. His implementation comes along with 28 stacks. [39] finally generalises these two concepts

to arbitrary dimensions, and she proves that these algorithms utilise $3^d + 1$ stacks[1]. In this dissertation, an alternative scheme coming along with $2d + 2$ stacks is proposed, although it does not pose any additional restrictions on the trees. I reduce the implementation complexity from exponential to linear.

The term space-filling is credited to the mathematicians Cantor, Netto, Peano and Hilbert, and its underlying ideas open the door to a rich, interesting field of research and theory. As many aspects of this theory are presented in [35, 39, 63], this chapter restricts to the construction and appropriate properties. A good survey on space-filling curves is [66]. Besides [35, 39, 63], there are additional applications of (alternative) space-filling curves not discussed here ([2, 32], e.g.).

The chapter is organised as follows: In Section 3.1, Peano space-filling curves, their construction principle, and some of their properties are introduced. The Peano curve's properties lead to the idea of stacks acting as data containers for the grid management. This management is described in Section 3.3, and, thus, this section holds the fundamental new contribution of the chapter. The following text reveals how today's hardware architectures and cache hierarchies benefit from the cache-based management, before some realisation details in Section 3.5 complete the chapter. These details on the one hand comprise the handling of different traversal depths as they occur for multiplicative multigrid algorithms. On the other hand, they transfer and extend a file-based stack realisation concept introduced by [63] to this work. With these file-based realisations, one can handle problems that need more main memory than actually available on the machine. Some experiments study the memory characteristics of the algorithm, and a short outlook closes the chapter.

3.1 Peano Space-Filling Curve

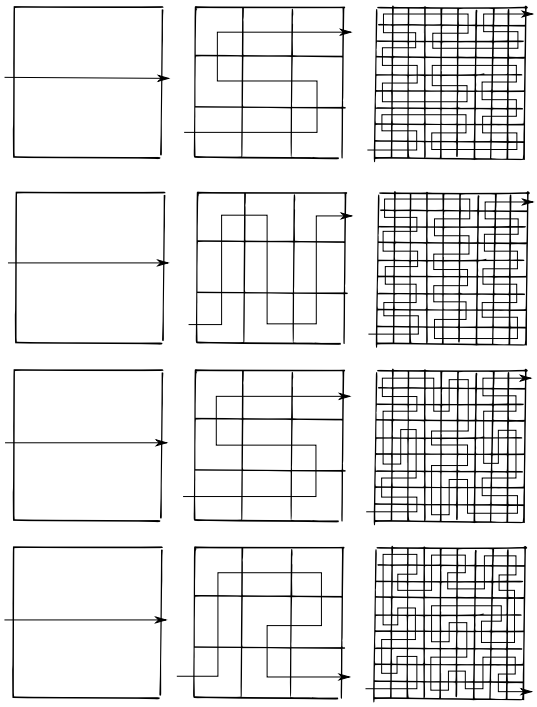
Peano's cache and memory efficiency rely on the construction principles and properties of the Peano space-filling curve. Although the principle of its construction can be understood analysing the illustrations in Figure 3.1, and although its three important properties formalised in the theorems on page 59 and the following are intuitively clear, a rigorous, closed formalism is important for stating subsequent algorithms. This section provides this formalism. It so leaves the concept of k-spacetrees aside, and concentrates on the concept of the space-filling curves. Skipping it, the ideas of Section 3.3 holding the grid storage and traversal algorithm remain plausible, but are neither provable nor re-programmable.

Space-filling curves are mathematical eccentrics whose existence and properties have been fiercely debated for a long time, as their character implies that the unit

[1] In all three theses, one can reduce the number of required stacks by 2^d with some straight modifications.

—Alternative Peano Curves—

In [66], the term Peano space-filling curve refers to a construction principle based upon three-partitioning of the unit square along each coordinate axis. Such a curve is neither unique with respect to rotation, nor is it unique with respect to the curve's layout. The following illustration gives four three times four iterates for four different Peano space-filling curves:



This thesis uses a standardised Peano space-filling curve (first row), where the curve always runs along the x_1 axis first, then x_2, then x_3, and so on. Another version of the curve changes the dominant traversal direction per iterate (second). The third variant exchanges the dominant traversal order for each second square. All three variants are of *switch-back type* (Serpentinentyp) exhibiting a $2\times1\times2\times1\times2\times1$ step pattern—two steps along one direction, then one step along an orthogonal direction. Finally, the fourth variant traverses each 3^2 motif with a $2 \times 2 \times 1 \times 1 \times 1 \times 2$ step pattern. This variant's original identifier is *Peano curve of the meander type*. The extension of all four variants to arbitrary dimension is straightforward, and the Austrian mathematician Walter Wunderlich coined the names.

square's volume equals the unit interval's volume. The term curve identifies a mapping from the unit interval to a higher-dimensional domain such as a square. The term space-filling denotes that the curve's image has a positive Jordan content: the image completely "floods" a higher-dimensional domain. This section presents the construction principle of one specific type of space-filling curves—the Peano curve.

The *Peano space-filling curve* is a surjective, continuous mapping from the unit interval to the unit hypercube. It is constructed recursively:

- The hypercube (image) is split up into 3^d equal subcubes.
- The unit interval (preimage) is split up into 3^d equal subintervals.
- Each subinterval maps to one subcube according to Figure 3.1, i.e. neighbour subintervals map to adjacent subcubes. The central illustration in Figure 3.1—the curve running through 3^d hypercubes—is the *leitmotiv*.
- Neighbouring subintervals' images are connected by the curve, and the curve defines an order on the subintervals.
- The curve's construction continues recursively for each subcube. For each subcube, the leitmotiv is mirrored accordingly: The connected leitmotivs fitted into the subcubes form a continuous curve, and this curve preserves the ordering of the coarser hypercubes resulting from the preceding construction step.

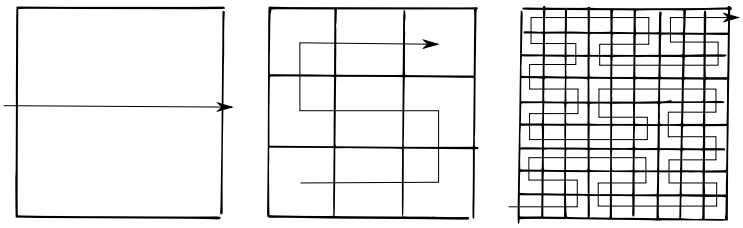


Figure 3.1: The first three iterates of the Peano curve for $d = 2$. For $d \to \infty$, the curve fills the unit square surjectively. The left hypercube corresponds to the construction scheme's recursion start, and the central illustration shows the leitmotiv. This leitmotiv defines the overall curve. The interconnections of the different suitable translated and mirrored leitmotivs on the right-hand side do not result from the leitmotiv directly, but result from the leitmotiv of the preceding recursion step.

With the recursion depth going to infinity, the curve completely fills out the whole unit hypercube. There are many variants of this space-filling curve based upon tripartitioning (see the excursus on page 54), and even more different mappings based

upon other partitioning techniques. The Hilbert curve based on bi-partitioning perhaps is the most popular one. Another example embedding the curve's image into triangles is the Sierpinski curve. However, the Peano curve has a number of unique properties which are very useful for a PDE solver realisation[2]. Before the subsequent subsections discuss these properties, the multitude of different Peano space-filling curves is worth a further attention.

The leitmotiv in Figure 3.1 equals a 2×1 pattern: From a subcube, the curve meanders two subcubes along one direction. Afterwards, it changes the direction orthogonally and proceeds one subcube. Although the curve's shape is thereby defined unambiguously, the overall curve's rotation is not fixed yet.

Definition 3.1. *A* standardised lexicographic *leitmotiv traverses a 3^d pattern always first along the x_1 axis. It then steps one subcube along the x_2 axis and, subsequently, continues to meander along the x_1 axis backwards. For $d \geq 3$, the next meander direction is x_3, then x_4 and so on.*

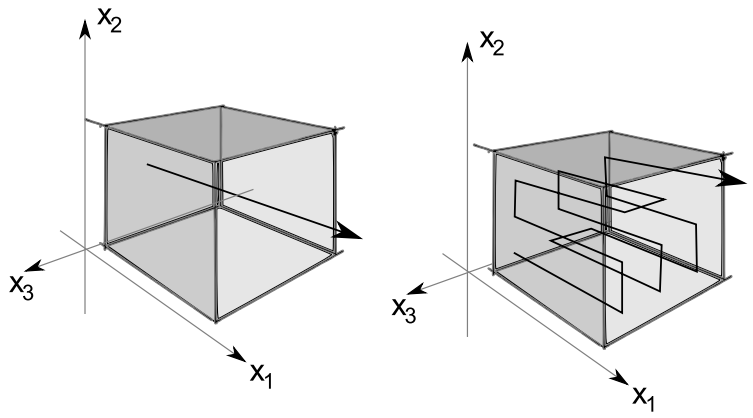


Figure 3.2: The first two iterates of the Peano curve for $d = 3$. On the right-hand side, the unit cube is cut into three plates and the curve runs through each plate according to the two-dimensional scheme. Afterwards, the three individual curve fragments are connected.

The term "mirrored accordingly" within the construction definition also demands for a more rigorous definition.

Definition 3.2. *In this thesis, the term* mirror along *or* in the direction of the x_i-axis *equals a reversion of the x_i-axis. For the unit hypercube, mirroring along the x_i-axis thus mirrors the curve at a hyperface with normal x_i.*

[2]This dissertation elaborates properties of the Peano curve based on tri-partitioning. Tri-partitioning then in turn fits to the $(k = 3)$-spacetree. However, the properties hold for any standardised meander-type curve corresponding to an odd k, i.e. the are of application is much wider (see outlook).

The "mirroring accordingly" is an affine mapping $\mathbf{P} = \mathbf{T}\hat{\mathbf{P}}\mathbf{T}$, where the generic operator $\mathbf{T}$ translates any hypercube to the unit hypercube or the other way round, and where $\hat{\mathbf{P}}$ performs the mirroring. The mirroring in turn depends on the hypercube's position within the preliminary construction step. One can express this position in terms of odd and even cubes along the coordinate axes. Let

$$even : subcube \mapsto \{\top, \bot\}^d$$

define a d-tuple on each subcube. It is the *even*-flag, and a subscript picks out a particular component of the image. *even* for the spacetree's root results in $\{\bot, \bot, \ldots, \bot\}$. If a cube is split-up, the first new subcube's *even*-flag along the iterate equals the original subcube's flag. For two subcubes a and b connected by a hyperface with normal x_i, $i \in \{0, \ldots, d-1\}$[3]

$$\begin{aligned} even_j(a) &= even_j(b) \qquad \forall j \neq i \qquad \text{and} \\ even_i(a) &= \neg even_i(b). \end{aligned} \tag{3.1}$$

holds (Figure 3.3). The function separates even from odd subcubes along each coordinate axis, and, thus, defines whether an element's iterate runs along the coordinate axis or not:

$$\begin{aligned} isTraversePositiveAlongAxis : \{\top, \bot\}^d \times \{1, \ldots, d\} &\mapsto \{\top, \bot\}, \qquad \text{with} \\ isTraversePositiveAlongAxis(even, axis) &= \top \quad \Leftrightarrow \\ \exists k \in \mathbb{N}_0 : |\{i : \ i \neq axis \wedge even_i = \top\}| &= 2 \cdot k. \end{aligned} \tag{3.2}$$

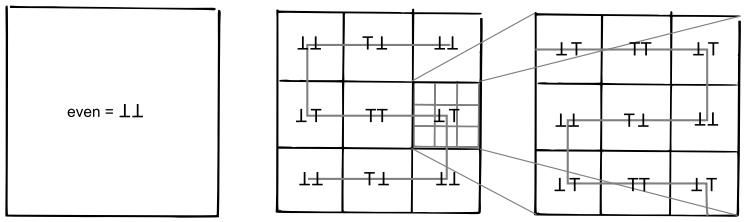


Figure 3.3: *even*-flag for a two-dimensional adaptive Cartesian grid. The flag uniquely determines how the leitmotiv is mirrored. Therefore, the flag also determines which neighbour square is crossed next by the iterate.

The function analyses the root node's leitmotiv: Here, the traversal runs from the cube point nearest to the origin to the point furthest away: It runs along each coordinate axis. *even* gives a d-dimensional module two enumeration, and it tracks how often the leitmotiv has been mirrored and translated. The translations do not affect the curve's shape. With this mirroring, there are four important observations:

[3] I enumerate all these tuples in C-style, i.e. starting with 0, to keep algorithms and the formalism consistent. Consequently, the coordinate system's axes are also enumerated in C-style.

- For the unit hypercube, the algorithm applies the standard lexicographic leitmotiv.
- For two hypercubes with the same *even*-flag, the algorithm applies the same leitmotiv.
- If two adjacent hypercubes' *even*-flags differ in one entry i, one leitmotiv results from the other cube's leitmotiv by mirroring it along each coordinate axis besides x_i.
- If two leitmotivs differ in more than one entry, the mirroring operations have to be applied consecutively. The image is deterministic and unique as the mirror operations are commutative.

A more formal description of the leitmotiv usage can be obtained by using a grammar to describe the recursion steps. Such a grammar can be found in [39] and [63], e.g. For this thesis, working with the *even*-flag is sufficient, as the individual algorithms exploit solely the traversal's orientation and neglect the transition within the affine operator.

Example 3.1. *For $d = 2$, the following mirroring operators arise.*

$$\hat{\mathbf{P}}_{\perp\perp} = id = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \hat{\mathbf{P}}_{\top\perp} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$
$$\hat{\mathbf{P}}_{\perp\top} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \hat{\mathbf{P}}_{\perp\perp} = \hat{\mathbf{P}}_{\perp\top}\hat{\mathbf{P}}_{\top\perp} = \hat{\mathbf{P}}_{\top\perp}\hat{\mathbf{P}}_{\perp\top} = -id.$$

For arbitrary d, the matrices evolve from

$$\hat{\mathbf{P}}_{\perp\perp\perp\ldots} = id,$$

$$\hat{\mathbf{P}}_{\underbrace{\perp\perp\perp\ldots\top\perp\ldots}_{i\text{th entry}}} = \begin{pmatrix} 1 & 0 & \ldots & & & & \\ 0 & 1 & \ldots & & & & \\ \ldots & & \ddots & & & & \\ & & & -1 & & & ith\ row \\ & & & & 1 & & \\ & & & & & \ddots & \end{pmatrix},$$

and the multiplication rule

$$\hat{\mathbf{P}}_{\underbrace{\perp\ldots\top\perp\ldots\top\perp\ldots}_{l\text{th and } k\text{th entry}}} = \hat{\mathbf{P}}_{\underbrace{\perp\perp\perp\ldots\top\perp\ldots}_{l\text{th entry}}} \hat{\mathbf{P}}_{\underbrace{\perp\perp\perp\ldots\top\perp\ldots}_{k\text{th entry}}}.$$

Since the involved matrices exhibit a diagonal pattern, they are commutative.

Each construction step applying the leitmotiv on all hypercubes of the smallest size yields a curve of its own. They are *iterates* of the Peano space-filling curve and belong to a non-linear recursion with order 3^d. The recursion's depth determines the level of the iterate. With these iterates and their construction blueprint at hand, the subsequent pages study the iterate's properties. Peano's spacetree traversal later imitates the curve's iterates and, thus, inherits these properties. They are in turn essential to construct the grid storage and traversal algorithm.

3.1.1 Projection Property

Theorem 3.1. *For the standardised Peano iterates, the* projection property *holds: Examine an iterate of level ℓ. A hyperplane is cut out from the unit hypercube along two hyperfaces with normal x_i. They are translated by $\frac{k}{3^\ell}$ and $\frac{k+1}{3^\ell}$ along the x_i-axis with a fixed $k \in \{0, \ldots, 3^\ell - 1\}$. The hyperplane contains a subcurve. If one projects the subcurve orthogonally to one of the cuts' hyperfaces, the image is in turn a $(d-1)$-dimensional Peano iterate.*

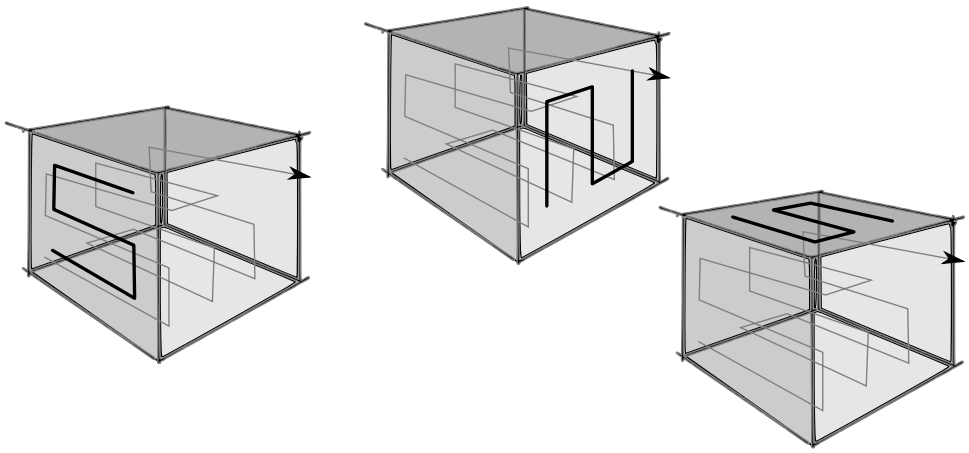


Figure 3.4: Projection property of the Peano space-filling curve for $d = 3$: If the iterate is mapped to a face of the unit cube, the image is in turn a $(d - 1 = 2)$-dimensional Peano iterate. This holds for all $2 \cdot d$ faces.

The proof results from the mirroring of the leitmotiv on page 57: The iterate within the cut plane corresponds to a sequence of leitmotiv transformations, and the following projection to the cut planes removes the i-th entry from the image. One can interchange construction as well as projection and ends up with a new construction scheme for a $d-1$-dimensional curve where the i-th component is eliminated. This construction scheme equals the $d-1$-dimensional Peano iterate. The proof is elaborated for example in [49], and the projection property itself is illustrated for $d = 3$ in Figure 3.4.

3.1.2 Inversion Property

Theorem 3.2. *For the standardised Peano iterates, the* inversion property *holds: Let i_a be the iterate belonging to level ℓ. If the leitmotiv of the unit hypercube is mirrored along each coordinate axis*[4]*, the construction scheme of the standardised Peano space-filling curve's definition also yields a Peano iterate i_b. i_a and i_b are congruent, but they have reverse orientations.*

Proof. The proof is a simple induction over the construction steps. For the induction start, the theorem holds, as the iterate's direction is inverted. For a refined event, the construction sequence of the 3^d subcubes is inverted. For each subcube the theorem holds by the induction. All the curves are then connected. The connection order equals the original curve's inverted arrangement. □

Corollary 3.1. *The set of Peano space-filling curves is* closed *under the invert traversal operation.*

3.1.3 Palindrome Property

Theorem 3.3. *For the standardised Peano iterates, the* palindrome property *holds: Let there be an iterate of level ℓ. Two neighbouring hyperplanes are cut out from the unit hypercube along three hyperfaces with normal x_i. The hyperfaces are translated by $\frac{k}{3^\ell}$, $\frac{k+1}{3^\ell}$, and $\frac{k+2}{3^\ell}$ along the x_i-axis with fixed $k \in \{0, \ldots, 3^\ell - 2\}$. Each hyperplane contains a subiterate i_a or i_b, respectively. i_a and i_b are congruent but have reversed orientations.*

Again, a formal proof is given by [49]. The property stems from the construction of the *even*-flag, as two hypercubes contained in different planes sharing one hyperface have an equal *even*-flag besides one entry corresponding to the hyperface's normal. If the two hyperplanes' iterates are mapped to the connecting hyperface, their mirroring operators thus differ in the sign, i.e. one leitmotiv results from the other by inverting the curve along each of the $d - 1$ coordinate axes (Figure 3.5).

3.2 Deterministic Peano Traversal for k-spacetrees

The construction of the Peano space-filling curve resembles the construction of a $(k = 3)$-spacetree. Both differ in three aspects: First, the grid generation stops after a finite number of recursion steps. The Peano curve instead results from the limit of the iterates. Second, the grid generation determines for each subcube individually

[4]Such a inversion equals the transition $\hat{\mathbf{P}} \mapsto -id \cdot \hat{\mathbf{P}}$ in the mirroring operator. For an even d, this transition in turn corresponds to an inversion of the even flags, i.e. $(\bot, \bot, \bot, \ldots) \mapsto (\top, \top, \top, \ldots)$, for the root element. For an odd d, this equivalence does not hold.

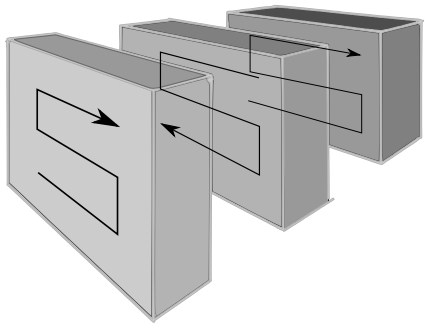


Figure 3.5: Palindrome property of the Peano space-filling curve: If the unit cube is cut into three plates along a coordinate axis, and the first Peano iterate is projected onto these plates (projection property in Figure 3.4), each iterate on the plate mirrors the neighbours' curves, i.e. it exhibits the same layout with a different traversal direction.

whether to refine further, and, thus, facilitates adaptive grids. The Peano iterates' image intervals equal a regular Cartesian grid. Third, the k-spacetree definition lacks a definition of an order of the children resulting from the recursion steps, whereas the Peano iterates define such an order on the image's interval.

Both depth-first and breath-first traversal on k-spacetrees are partially non-deterministic: They obey to the order $\sqsubseteq_{child}$, as they exploit the tree structures. Yet, the loops in the traversal algorithms do not arrange the children. It is an obvious idea to use the Peano space-filling iterates to derive such an arrangement. The resulting spacetree traversal then inherits all the properties of the Peano space-filling curve or its iterates, respectively. Due to the projection property, algorithms exhibit a tensor product style. Due to the inversion property, the traversal can toggle its orientation after each run through the spacetree, and, nevertheless, it preserves its behaviour. And the palindrome property finally is the missing link to realise the data containers with stacks.

Definition 3.3. *A* Peano spacetree *is a* $(k = 3)$*-spacetree with a deterministic traversal preserving the child-father relationship. Furthermore, the siblings' order first preserves* $\sqsubseteq_{\text{pre}}$. $\sqsubseteq_{\text{pre}}$ *in turn is the order induced by the leitmotiv corresponding to the parent's even-flag. The* $(k = 3)$*-spacetree's root corresponds to an arbitrary event flag.*

The depth-first traversal for a two-dimensional Peano spacetree is illustrated in Figure 3.6. It is realised recursively by a stack automaton (usually represented by the call stack), and it benefits from the definition of the *even*-flag: The automaton holds—besides spatial location, level, and so forth—the *even*-flag of the current geometric element. If one adjacent vertex holds $\mathcal{P}_{\text{refined}}$, the automaton has to descend.

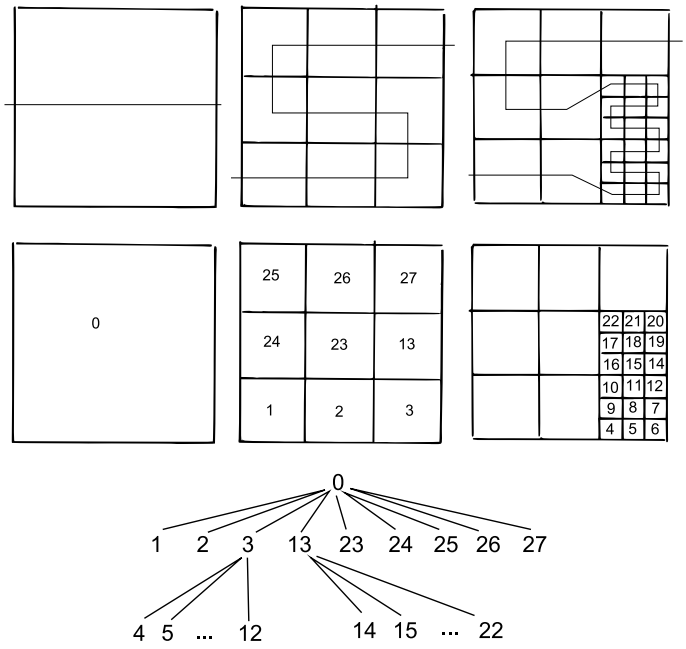


Figure 3.6: $(k = 3)$-spacetree with a depth-first ordering derived from the iterates of the Peano space-filling curve. The first row illustrates the Peano iterate on the different levels. The second row gives the enumeration corresponding to the depth-first search. At the bottom, the corresponding space-tree is illustrated.

The *even*-flag identifies the first child into which to descend, and the *even*-flag also determines the order of the subsequent descends due to the leitmotiv.

The counterpart of the inversion property for the stack automaton is the following definition:

Definition 3.4. *The image of the* invert traversal operation

$$invert : \mathcal{T} \mapsto \mathcal{T}$$

on a Peano spacetree is a tree with the same geometric elements, with the same vertices, and with the same root element,

- *where the leitmotiv's direction is inverted,*
- *where the same child-father relationship* $\sqsubseteq_{child}$ *holds*

$$\forall e_i \sqsubseteq_{child,\mathcal{T}} e_j : e_i \sqsubseteq_{child,invert(\mathcal{T})} e_j,$$

- *but where the siblings' order* $\sqsubseteq_{\text{pre}}$ *is inverted:*

$$\forall e_i, e_j \sqsubseteq_{child,\mathcal{T}} e_k : e_i \sqsubseteq_{\text{pre},\mathcal{T}} e_j \Leftrightarrow e_j \sqsubseteq_{\text{pre},invert(\mathcal{T})} e_i.$$

Corollary 3.2. *The set of Peano spacetrees is* closed *under the invert traversal operation, i.e. the image is in turn a Peano spacetree.*

Proof. The proof results from the inversion property. □

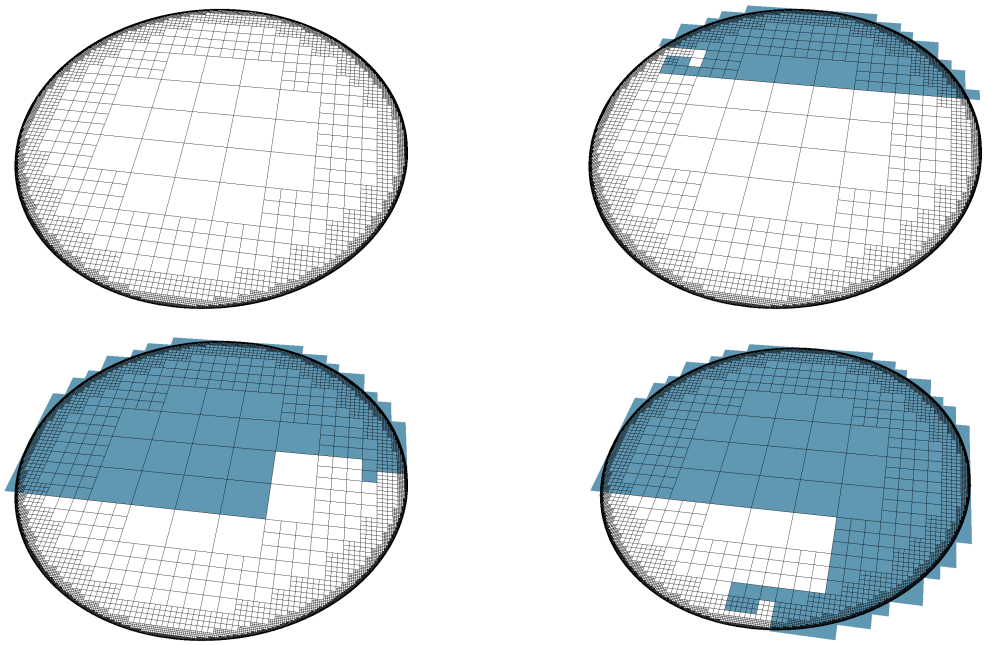


Figure 3.7: The Peano space-filling curve's iterate runs through an adaptive $(k = 3)$-spacetree discretising a circle. Adaptive grid with three snapshots of a running traversal. All fine grid elements already visited are inked.

The deterministic stack automaton sketched above, a definition of the geometry representation, and the refinement transition concept deliver a blueprint for a deterministic traversal algorithm on dynamic adaptive Cartesian grids resulting from Peano spacetrees. Yet, this blueprint lacks the description of containers for data associated to vertices or elements.

3.3 Stack-Based Containers

Space-filling curve and depth-first traversal in combination shape the deterministic Peano spacetree traversal. To implement this traversal, a programmer has to combine this simple traversal blueprint with fitting containers for vertices and elements. The data flow and data access pattern underlying a k-spacetree traversal define the access operations the data containers have to provide. Due to the properties of

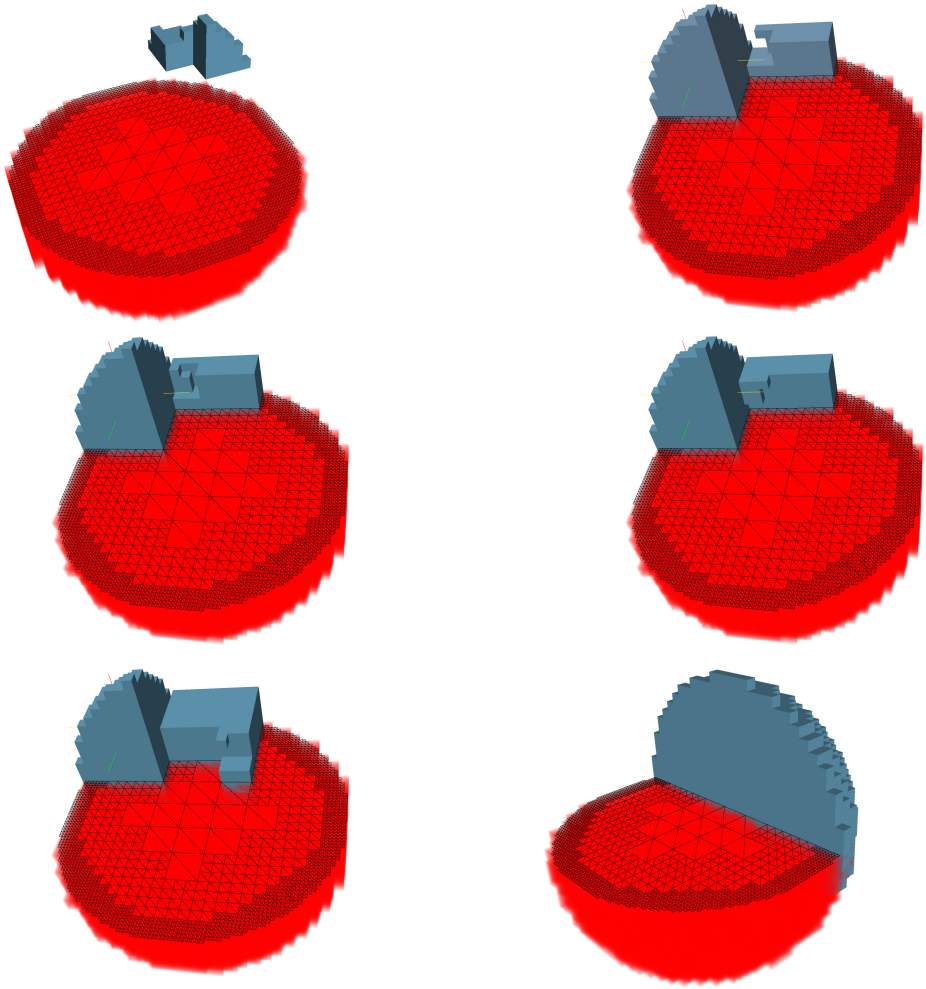


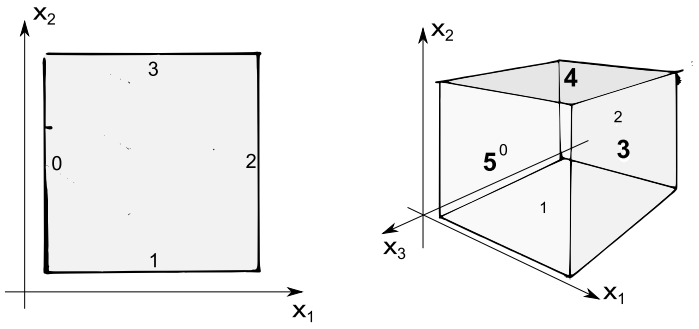
Figure 3.8: The Peano space-filling curve's iterate runs through an adaptive ($k = 3$)-spacetree discretising a sphere. Six snapshots of a running traversal. All fine grid elements already visited are inked.

space-filling curves and their iterations, the operations comprise solely operations from a stack signature, while the data access patterns for vertices and geometric elements exhibit a different behaviour. The section at hand elaborates this behaviour. It starts with the element data, continues with the storage of vertex data in-between two iterations, and ends up with the stacks required throughout the traversal itself. As a result, it presents a spacetree traversal realisation and a storage scheme coming along with a small, fixed number of stacks as data containers.

—Face Enumeration—

Each geometric element has $2 \cdot d$ faces. These faces' enumeration is deduced from the faces' normal as follows:

1. The numbers 0 and $0 + d$ identify the faces with normal x_1.
2. The numbers 1 and $1 + d$ identify the faces with normal x_2.
3. The numbers 2 and $2 + d$ identify the faces with normal x_3.
4. ...



With two faces having the same normal, the face closer to the coordinate system's origin is enumerated first.

3.3.1 Container for the Geometric Elements

Every time a geometric element is read for the first time, its data is taken from an input container, the corresponding vertices are read, and the traversal triggers the corresponding event *enterElement* (Table 2.2). Afterwards, the traversal descends recursively if the current geometric element is refined. In this case, the call stack holds both the traversal's state (position, element width, and so forth) represented by a traversal automaton as well as the associated grid data (vertices and data assigned to the geometric elements) until the recursion terminates. Finally, the algorithm triggers *leaveElement*, deposits the vertices, and writes the element to an output container. This write has a fire-and-forget semantics, i.e. each element written is not read again throughout this traversal. For the geometric elements, there is thus an input and an output stream.

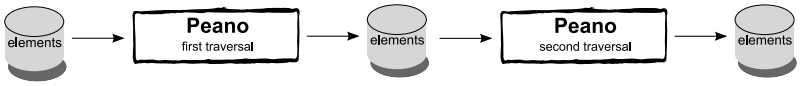
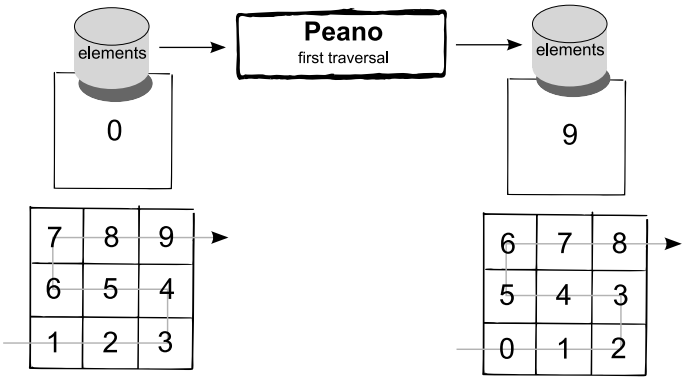


Figure 3.9: The traversal algorithm reads geometric elements from an input stream, and stores the record to the output stream. The output stream then acts as input stream for the second traversal.

For two subsequent traversals, the records stored on the output stream of the first iteration act as input for the second iteration (Figure 3.9). The order on the corresponding output stream has to equal the order of the input stream. Yet, the order of the elements on the input and output stream is not equal for a straightforward depth-first type traversal:

Example 3.2. *Examine a spacetree of height one. The refined root element is read before the 3^d children are read. When the traversal descends, the parent element is held on the call stack. As the children are leaves, they are hence written to the output stream in the same order in which they are read. Finally, the parent geometric element is written to the output stream:*



In a depth-first traversal, parent nodes are always read before their children. In turn, they are written to the output stream after their children have been written.

Although input and output order do not concur, it is oblivious that the resorting equals an inversion of the storage order. An algorithm switching the read access order for each traversal thus comes along without any resorting of the containers holding the geometric elements.

Theorem 3.4. *Each Peano traversal defines an order in which the geometric elements are read (input order) and an order in which the geometric elements are written (output order). The elements' output order for a Peano spacetree $\mathcal{T}$ equals the inverse input order for the Peano spacetree $invert(\mathcal{T})$.*

Proof. The proof is a simple induction over the height of the Peano spacetree. The theorem holds for a spacetree of height zero, i.e. one single hypercube. For a refined geometric element $e \in inverse(\mathcal{T})$, the leitmotiv is mirrored along every axis and

$$\forall e_i, e_j \sqsubseteq_{\text{child}} e, e_i \sqsubseteq_{pre,\mathcal{T}} e_j : e_j \sqsubseteq_{pre,inverse(\mathcal{T})} e_i. \tag{3.3}$$

$\mathcal{T}$'s traversal writes e as last element of the output stream. $inverse(\mathcal{T})$'s traversal reads e first. According to (3.3), the last geometric element e_1 written to the output by the traversal is the the first geometric element read by the inverse traversal. For the siblings of e_1, the theorem holds due to the induction. The subsequent child e_2 of e read by the inverse traversal is the element written by the traversal before it reads and descends into e_1. For the siblings of e_2, the theorem holds due to the induction. The arguing is continued for e_3 up to e_{3^d}. □

The theorem does not have to discuss dynamic refinement, as it does not correlate the input and output order of the same traversal: Adding or removing spacetree elements enrich or downsize the output stream, but do not alter the stream access pattern or the stream's semantics.

Running through a stream once in one direction then in the opposite direction equals a sequence of push operations followed by a sequence of pop operations:

Corollary 3.3. *The input and output containers for geometric elements are* stacks.

While I implement the containers as stacks, this realisation is far too general as it does not take into account the stream characteristics: The two stacks are always emptied completely before a sequence of push operations refills them. More sophisticated implementations could exploit this insight and, due to a reduced functionality, provide an optimised container realisation. The stream access pattern permits implementations for example to deploy data pre- and postprocessing to a thread of their own [23].

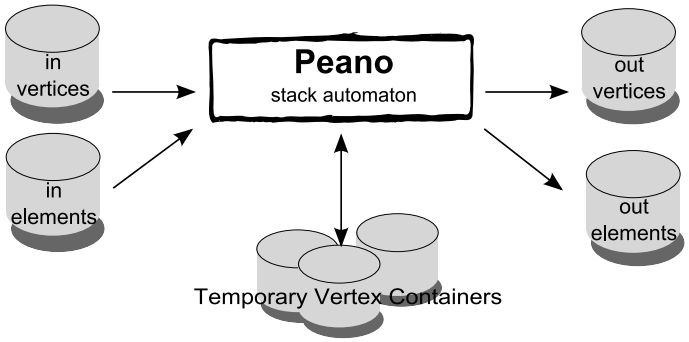


Figure 3.10: Throughout the traversal, the algorithm reads vertex and element data from two input streams. If a record is not needed anymore, the record is written to an output stream. In between, records for geometric elements are stored on the call stack. Vertices, however, are manipulated by several geometric elements. Thus, they have to be stored within temporary containers after the first read, until they have been used by all 2^d adjacent geometric elements.

3.3.2 Input and Output Container for Vertices

The Peano spacetree's traversal equals a resorting of the geometric elements: From a data flow point of view, it reads elements from an input stream and writes them to an output stream in a permuted order. The data flow for the vertices is more complicated, as vertices are processed 2^d times by the element-wise traversal—once per adjacent element. A vertex is read from an input stream, too, but it is not written to an output stream immediately. The algorithm instead reads a vertex from an input stream when the first adjacent geometric element is entered. Afterwards, it has to store the vertex within a temporary container. After it has been read or written, respectively, from or to the temporary container $2^d - 1$ times, i.e. the total number of read operations equals 2^d, the algorithm finally stores the vertex in an output container (Figure 3.10). This subsection discusses the input and output container.

The vertex transition scheme in Section 2.6 ensures that each vertex $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ is processed 2^d times. Hanging vertices $v \in \mathbb{H}_{\mathcal{T}}$ are created on-the-fly, and they are not stored on the input and output streams. Because of the vertex state transition scheme, the refinement and coarsening is completely transparent: The algorithm does not distinguish between existing vertices and new vertices. Existing vertices are read from the input stream, while new vertices are created on-the-fly. Besides the input stream there is thus a vertex source. If elements are to be destroyed due to coarsening the algorithm does not store these vertices on the output stream but throws them away. The same holds for hanging vertices. Besides the output stream, there is thus a vertex sink.

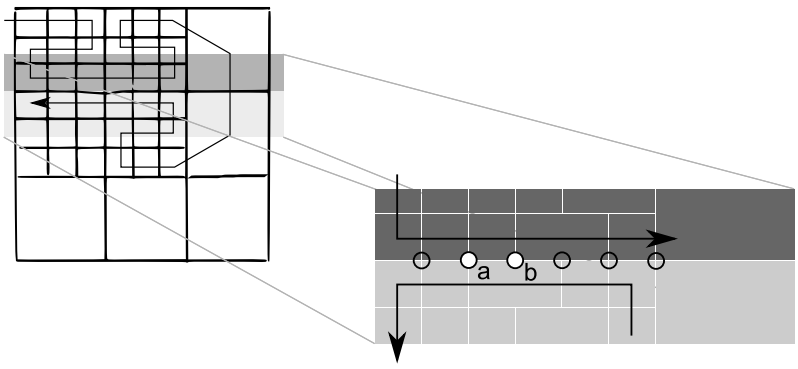


Figure 3.11: The Peano iterate runs along the vertices a and b first from left to right (top) and then from right to left (bottom). This behaviour results from the palindrome property. Since a non-hanging, i.e. persistent, vertex is written to the output stream as soon as it has been read by all 2^d adjacent vertices, vertex a is read from the input stream before b and is written to the output stream after vertex b is written. If the traversal direction is inverted after the iteration, vertex a again is read before vertex b.

Theorem 3.5. *Each Peano traversal defines an order in which the vertices are read (input order) from the input stream, and it defines an order in which the vertices are finally written (output order). The vertices' output order for a Peano spacetree $\mathcal{T}$ equals the inverse input order for the Peano spacetree invert($\mathcal{T}$).*

Proof. The proof for one single grid level derives from the inversion property. For the traversal of the whole spacetree, the statement holds by induction, as the traversal preserves the child-father relationship as well as the inverse child-father relationship (Definition 2.2). □

Corollary 3.4. *The input and output containers for vertices are* stacks.

Both the vertex and the element input and output containers are stacks and fit to an algorithm switching the traversal direction after each iteration. Again, stacks even are a too generalised data structure, as they are accessed in a stream manner. The number of grid levels does not influence the realisation, and the implementation of the depth-first traversal in Algorithm 2.4 thus is straightforward. Nevertheless, it relies on a local criterion deciding whether to read a vertex from the input stream or a temporary container. The same argumentation holds for the decision where to write the vertex to (Algorithm 3.1).

If a stack automaton realises the element-wise traversal of the spacetree, i.e. it runs through the geometric elements, it represents the individual states stored on the call stack. Each geometric element has $2 \cdot d$ faces. As each traversal state

Algorithm 3.1 The depth-first traversal traversal from Algorithm 2.1 without the events. Input and output streams are stacks. The temporary vertex container's realisation is not specified yet, and the for loop has to be made deterministic with the Peano curve.

```
 1: procedure dfs(e)
 2:     ...
 3:     for e_i ⊑_child e do
 4:         pop geometric element e_i from element input stream
 5:         for v_j ∈ vertex(e_i) do
 6:             if first read of v_j then                    ▷ see Algorithm A.1
 7:                 pop v_j from vertex input stream
 8:             else
 9:                 read v_j from temporary vertex container
10:             end if
11:         end for
12:         DFS(e_i)
13:         push geometric element e_i on element output stream
14:         for v_j ∈ vertex(e_i) do
15:             if v_j read 2^d times then                   ▷ see Algorithm A.1
16:                 push v_j on vertex output stream
17:             else
18:                 write v_j to temporary vertex container
19:             end if
20:         end for
21:     end for
22:     ...
23: end procedure
```

corresponds to one geometric element, each automaton state "has" $2 \cdot d$ faces. The enumeration of its faces is defined on page 65. Let $\mathcal{P}_{\text{touched}} : \{0, \ldots, 2d-1\} \mapsto \{\top, \bot\}$ declare whether the traversal automaton has already traversed the geometric element connected by a face. $\mathcal{P}_{\text{touched}}$ is the *touched predicate*. It depends exclusively on the stack automaton's state and is formalised later. The predicate is $\bot$ for all the faces of the root element. If an element is refined, the iterate's orientation (i.e. the mirroring of the leitmotiv) derives for all the 3^d recursion steps and all the $2 \cdot d$ faces per recursion step whether $\mathcal{P}_{\text{touched}}$ holds[5].

With $\mathcal{P}_{\text{touched}}$ assigned to each geometric element in combination with the traversal

[5] Although the mirroring of the leitmotiv, i.e. the *even* flags, determines when $\mathcal{P}_{\text{touched}}$ in the subsequent recursion steps holds, $\mathcal{P}_{\text{touched}}$ and *even* are not equivalent, as $\mathcal{P}_{\text{touched}}$ also depends on the father element's touched predicate.

automaton (it depends on the traversal direction, too), the traversal automaton can determine where to read a vertex from and where to store a vertex. If no face adjacent to a vertex holds the $\mathcal{P}_{\text{touched}}$ flag, the algorithm pops the record from the input stack. Otherwise, it reads from the temporary container. If all faces hold $\mathcal{P}_{\text{touched}}$, the algorithm pushes the record to the output stack. Otherwise, it passes the record to the temporary container. I present the algorithm deriving $\mathcal{P}_{\text{touched}}$ later, as it turns out that $\mathcal{P}_{\text{touched}}$ evaluates a more general automaton property needed anyway for the temporary stack containers. Algorithm A.1 comprising two helpers formalises the logic described above.

3.3.3 Temporary Stack Container

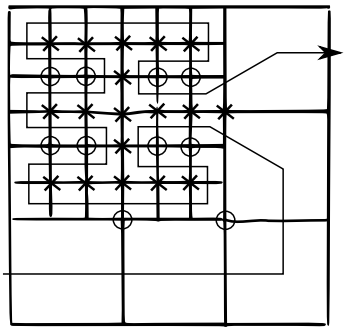


Figure 3.12: The continuous iterate splits up the vertices into vertices left of the iterate (circles) and right of the iterate (crosses).

The realisation of the input and output containers is more or less trivial, i.e. having stacks storing streams is not surprising. The subsequent text reveals that stacks are in fact the only data structure needed for the complete grid storage.

Since the Peano curve's iterate meanders through the discretised computational domain and passes through each geometric element once, it transits from one element into a neighbouring element through a common face. And as the construction principle yields continuous iterates, the discrete domain's vertices in $d = 2$ either are on the right-hand side of the curve or on the left-hand side (Figure 3.12). This holds for all levels.

Let vertex a and vertex b on level ℓ belong to the class of left-hand side vertices ($d = 2$). For the continuity of the iterate and the palindrome property, the following facts hold:

- If a is read from the vertex input stream before b, a is stored on the temporary vertex container for the first time before b.
- Afterwards, b will be read before a, i.e. the second vertex read order is inverted.

Algorithm 3.2 In each geometric element, the vertex load order has to fit to the traversal's direction. The for loop in Algorithm 3.1 thus has represents the following loop—the loop's body instructions are to be embedded at line 13. The code snippet evaluates the automatons current *even* flag fixing the iterate's direction. The source fragment is to be optimised, but the non-optimised version enlights the traversal mirroring. $mask$ and $firstVertex$ are modified in their binary representation, and $firstVertex$ represents a vertex position within one element, i.e. as a $(0,1)^d$ tuple.

```
1:  firstVertex ← (0,0,...); mask ← (0,0,...)
2:  for i ∈ {0, d−1} do
3:      if even_i then
4:          mask_i ← 1
5:          mask ← ¬mask
6:          firstVertex ← firstVertex ⊻ mask
7:          mask ← ¬mask
8:          mask_i ← 0
9:      end if
10: end for
11: for i = 0 : 2^d do
12:     currentVertex ← firstVertex ⊻ i
13:     ...
14: end for
```

As the access order inverts, it is an obvious idea to use again stacks as temporary containers, and to extend this concept to arbitrary dimensions. To make this approach work, the sequence in which the vertices are loaded for an element has to fit to the space-filling curve, too. Thus, the two loops "**for** $v_j \in vertex(e_i)$ **do** " in Algorithm 3.1 fit to Algorithm 3.2.

A Left/right Classifier

The fundamental idea of [35, 39, 63] is to store all the vertices left of the iterate on one stack and all the vertices right of the iterate on another stack. There is a left and a right stack. If the traversal automaton leaves a geometric element, the algorithm swaps all vertices that are to be read again to one of the two stacks according to the order induced by the iterate running through the element. The iterate within the neighbouring element is inverted due to the palindrome property, i.e. in a neighbouring element the automaton reads the temporary vertices in reversed order—it takes the vertices from a temporary stack:

- The operation "**read** v_j from temporary vertex container" in Algorithm 3.1 becomes "if v_j is left of iterate, **pop** it from the left stack, otherwise **pop** it from the right stack".

- The operation "**write** v_j to temporary vertex container" in Algorithm 3.1 becomes "if v_j is left of iterate, **push** it to the left stack, otherwise **push** it to the right stack".

As the projection property holds for the Peano curve, this idea generalises to any d via induction.

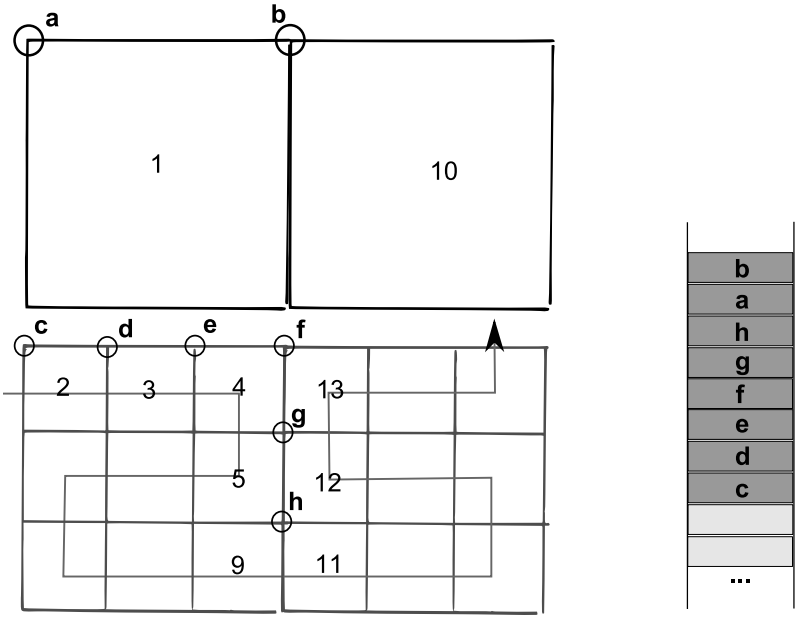


Figure 3.13: Only a left and a right temporary stack does not work for the Peano spacetree traversal: In the example, the left stack is studied. The geometric elements $2, 3, 4, 5, 9$ swap the vertices **c**,**d**,**e**,**f**,**g** and **h** to the left stack. Afterwards, the traversal algorithm ascends into the parent element, and the vertices **a** and **b** are written to the left stack. The algorithm then transits from element 1 into element 10. Within 10, **b** is taken from the left stack and the traversal descends into the first child (11). This child has to read **h** from the left stack. Yet, vertex **a** is on top of the stack and shadows the vertices **f**, **g** and **h**. The two-stack algorithm does not work.

The Peano spacetree traversal runs through all levels of the spacetree. Unfortunately, a simple left/right temporary stack concept then results in a container access conflict (Figure 3.13). Two approaches preserving the stacks-only policy resolve this conflict: Either each grid level holds a pair of left/right stacks of its own. Or a small, and, in particular, resolution independent, fixed number of additional "hierarchy" stacks locally resolves the access conflicts resulting from the multilevel interplay.

The first solution is straightforward but renders the stack idea null and void, as it introduces a dynamic data structure mapping levels to pairs of stacks. This

mapping then establishes an additional indirect memory access for a lookup table, and the table itself introduces an overhead as it grows with the maximum spacetree depth. Finally, the traversal permanently switches from one level to another, i.e. it would access different stacks permanently. Such an access behaviour leads to bad cache hit rates—an argument picked up later. Hence, [35, 39, 63] follow the other idea: The shadowing problem from Figure 3.13 affects vertices on the faces, i.e. the problem corresponds to a submanifold. They hence introduce additional stacks corresponding to the submanifold and write the vertices causing problems (vertex **a** in the example) to these additional submanifold stacks. The exact rules where to place which vertices become more complicated, but an induction over d and the grid hierarchy shows that the resulting algorithm succeeds with a fixed number of $3^d + 1$ stacks, i.e. the number of stacks does not depend on the spacetree's depth.

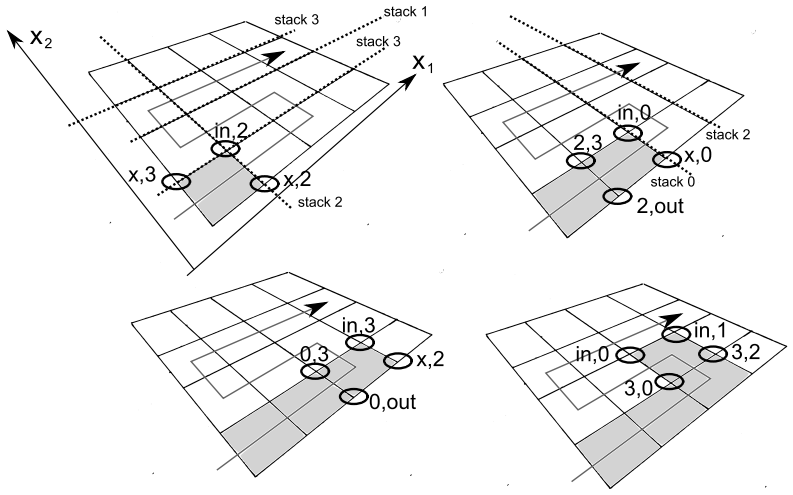
Alternative Concept

I derive an alternative access pattern for the temporary stack containers. Here, the traversal automaton keeps track of the order in which the neighbours are connected by a hyperface: Due to the standardisation of the Peano iterate in Definition 3.1, the stack automaton knows for each element connected by a face whether it will be visited or has been visited. For each vertex, the Peano traversal will next access the vertex out of a neighbour connected by a hyperface—not an edge, not a vertex, but a face. This is due to the continuity of the curve. Each automaton state thus knows, which neighbouring element triggers the next read access. It also knows which neighbouring element has processed a vertex before.

Let there be two stacks per dimension. They are enumerated oscillatingly, similar to the *even*-flag. Then, each element's face corresponds to a different stack due to the fact that there is an even number (two) of stacks but an odd number of partitions ($k = 3$) per coordinate axis.

Preceding a write to the temporary containers, the algorithm identifies the face connecting the element triggering the next read access. This face has a stack number. The algorithm pushes the element onto the stack with this number. The read operation then processes the counterpart of this analysis, and the overall algorithm comes along with $2d$ temporary stacks.

Example 3.3. *Given is a two-dimensional grid with the stack numbers* $0, 1, 2$ *and* 3*. In the following illustration, each tuple denotes the read stack (first entry) and the write stack (second entry).* x *denotes an unknown stack (not important in the example), and* in *and* out *are the input or output stacks respectively.*



In the first element (top, left), the traversal runs along the x_1 axis. Thus, two vertices are stored on stack 2, the third entry is stored on stack 3, as it will be used by the element along the x_2 axis next. In the second element (top, right), the traversal automaton reloads the vertices from stack 2. Although the traversal also continues along the x_1 axis (bottom, left), the vertices adjacent to the next element are stored on stack 0. This is due to the stack identifiers alternating along each coordinate axis.

It is obvious that this approach works for one single level, although it requires twice the number of stacks compared to a left/right classification. Yet, the shadowing problem sketched in Figure 3.13 can not occur, as the algorithm a priori swaps the vertices to the stacks from which they are needed next. The approach hereby relies on a fact mentioned above: Each pair of opposite faces belongs to different stack numbers.

Corollary 3.5. *The temporary container consists of $2d$ stacks.*

A formal proof is beyond the scope of this work, Yet, I formalise the underlying algorithm in more detail. Let

$$access : \{0, \ldots, 2d-1\} \mapsto \{-2d+1, -2d+2, \ldots, 0, 1, \ldots, 2d-1\} \tag{3.4}$$

define a value for the faces of a geometric element. The semantics of *access* is as follows:

$access(f) < 0$	Neighbour connected via face f has been processed before element was entered.
$access(f) > 0$	Neighbour connected via face f has not been processed yet, and will be entered after element has been left.
$access(f) = 0$	Face f is covered by the root element's faces.
$access(f) = -1$	Element has been entered through face f.
$access(f) = 1$	Element is left through face f.
$access(i) = m$	Neighbour connected via face i will be processed after all neighbours connected via face j for which $access(j) < m$ holds.

The constraints

$$\begin{aligned} access(i) > 0 \quad &\Rightarrow \quad \forall\ 0 < m \leq access(i), \exists j : access(j) = m, \\ access(i) < 0 \quad &\Rightarrow \quad \forall\ access(i) \leq m < 0, \exists j : access(j) = m, \quad \text{and} \qquad (3.5) \\ &\qquad\quad \forall i, access(i) \neq 0, \nexists j : access(j) = access(i) \qquad (3.6) \end{aligned}$$

ensure that no numbers within one access list are left out, i.e. no natural number is omitted. Furthermore, the numbers resulting from *access* are unique (3.6). The predicate $\mathcal{P}_{\text{touched}}$ then is

$$\begin{aligned} access(f) < 0 \quad &\Rightarrow \quad \mathcal{P}_{\text{touched}}, \\ access(f) >= 0 \quad &\Rightarrow \quad \neg\mathcal{P}_{\text{touched}}, \end{aligned}$$

and the range of the *access*'s image is bounded and fulfills (3.4). The spacetree's root element corresponds to

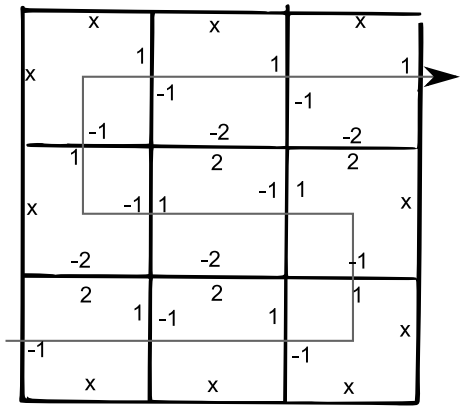
$$access(f) \quad = \quad 0 \quad \forall\ f \in \{0, \ldots, 2d-1\}.$$

In accordance with the *even* flag, a *access* flag also is given by the Peano iterate's construction. Whenever the automaton descends from a refined event into a child, it performs the following steps:

- Clone the parent automaton state.
- Derive the new *even* attributes.
- Identify the new entry face f_{entry}, set *access* value of f_{entry} to -1, and adopt other access entries such that (3.5) and (3.6) hold again.
- Identify the new exit face f_{exit}, set *access* value of f_{exit} to 1, and adopt other access entries such that (3.5) and (3.6) hold again.

A formal description of these steps is given in Algorithm A.2, Algorithm A.3, Algorithm A.4, and Algorithm A.5 in the appendix.

Example 3.4. *The access flags for the geometric elements in Example 3.3.*



The access flags denoted by x depend on the parent element's access flags.

To validate that each new state fulfills the stated constraints is trivial and not done here. The identification of the temporary stack now is as follows (Algorithms 3.3 and 3.4): If a vertex is to be read and all adjacent face's *access* entries are positive, it has to be taken from the input stream ($\mathcal{P}_{\text{touched}}$ does not hold for any adjacent face). Otherwise, one has to select f with the biggest negative $access(f)$ value among the adjacent faces. f identifies a stack, and f also corresponds to an element's face normal. If the *even* flag of the state along this normal does not hold, f is the temporary stack number searched for. Otherwise, f does identify the stack that belongs to the face that is opposite to the face belonging to the stack searched for.

If a vertex is to be written and all adjacent face's *access* entries are negative, it has to be written to the output stream. Otherwise, one has to select f with the smallest positive $access(f)$ value among the adjacent faces. f again identifies a stack and corresponds to a normal. If the *even* flag for this normal does not hold, f is the temporary stack number searched for. Otherwise, f does identify the stack that belongs to the face that is opposite the face to which the stack searched for does belong to.

The alternative stack management presented here is—from my point of view—a significant improvement compared to [35, 39, 63] for two reasons: It reduces the number of stacks for $d \geq 3$ and the grid hierarchy does not affect the vertex access, i.e. the information where to store a vertex or where to take a vertex from does not depend on the fact whether the vertex's position coincides with a vertex of a smaller level. The first aspect gains weight with increasing dimension d.

Peano offers a data persistence layer comprising exclusively of stacks, and the element-wise grid traversal is well-defined, too, according to this chapter. Before I switch to a concrete application built on top of the traversal due to a mapping from

Algorithm 3.3 Determine temporary stack number to read vertex at position from. If operation returns `UseInputStream`, vertex is to be read from input stream instead of a temporary stack.

$$getReadStack : \{0,1\}^d \mapsto \{0,\dots,2d-1\} \cup \{\texttt{UseInputStream}\}$$

```
1: procedure getReadStack(position)
2:     smallestValue ← −2 · d − 1
3:     result ← UseInputStream
4:     direction ← −1
5:     for i ∈ {0, d − 1} do
6:         if position_i = 0 then
7:             face ← i
8:         else
9:             face ← i + d
10:        end if
11:        if access_face < 0 ∧ access_face > smallestValue then
12:            result ← face
13:            smallestValue ← access_face
14:            direction ← i
15:        end if
16:        if result ≠ UseInputStream ∧ even_direction = ⊤ then
17:            if result < d then
18:                result ← result + d
19:            else
20:                result ← result − d
21:            end if
22:        end if
23:    end for
24:    return result
25: end procedure
```

Algorithm 3.4 Determine temporary stack number to write vertex at position to. If operation returns `UseOutputStream`, vertex is to be written to output stream instead of a temporary stack.

$$getWriteStack : \{0,1\}^d \mapsto \{0,\ldots,2d-1\} \cup \{\texttt{UseOutputStream}\}$$

```
1: procedure getWriteStack(position)
2:     biggestValue ← 2 · d + 1
3:     result ← UseOutputStream
4:     direction ← −1
5:     for i ∈ {0, d − 1} do
6:         if position_i = 0 then
7:             face ← i
8:         else
9:             face ← i + d
10:        end if
11:        if access_face > 0 ∧ access_face < biggestValue then
12:            result ← face
13:            biggestValue ← access_face
14:            direction ← i
15:        end if
16:        if result ≠ UseOutputStream ∧ even_direction = ⊤ then
17:            if result < d then
18:                result ← result + d
19:            else
20:                result ← result − d
21:            end if
22:        end if
23:    end for
24:    return result
25: end procedure
```

—von-Neumann Bottleneck—

A fundamental principle in computer construction is to distinguish between memory and processing units among other components. This idea goes back to János Lajos Neumann (John von Neumann) [74]. The processing unit's speed then determines the theoretical computing power of a system—the peak performance. The memory's size determines the maximum amount of data an application can handle, as long as no additional swapping techniques are applied. Both components are connected: The processing unit has to fetch data from the memory and write data back, as its local memory (registers) is extremely small compared to the main memory. Nowadays, this connection typically is a bus.
For such systems, the actual performance of an application does not depend exclusively on the peak performance anymore: Whenever data is to be read from the memory, the minimum of the memory's, the bus' and the processing unit's performance determines the actual execution speed. The first two aspects go back to the latency and bandwidth of the memory connection. Applications restricted by these effects are called *memory bounded*, and, in accordance with [74], people often speak of the von-Neumann bottleneck, if the bus causes the slowdown.

events to concrete operations, I discuss some properties arising from the interplay of space-filling curves and stacks.

3.4 Cache Efficiency

A stack-based grid traversal and storage scheme for adaptive Cartesian multiresolution grids is an interesting subject of study from an algorithmic point of view. The unique selling point of the approach is the low memory requirements mentioned several times. Another nice property stems from the exclusive use of stacks: The cache hit rate of the algorithm is extraordinary good as the following section points out. As a result, the framework was never memory-bounded in my experiments—a surprising and promising property for a PDE solver framework, in particular for applications typically running into the von-Neumann bottleneck.

How an algorithm exploits the memory hierarchy of an architecture determines whether the algorithm is cache efficient. There are many approaches to make an algorithm cache efficient, and there are many approaches to classify cache efficient algorithms. A fundamental classification distinguishes between *cache-aware* and

—Caches in the Memory Hierarchy—

Many applications are memory bounded, i.e. they do not exploit the processing unit's power, as the processing unit permanently has to wait for the memory system to deliver data. This problem becomes more and more burdensome: On the one hand, the processing units get faster due to an increased frequency or an increased number of cores. On the other hand, the memory becomes bigger, and this grow slows down the memory access speed.
To overcome the memory bottleneck, computer architects introduce a hierarchical memory system. In-between the processing unit and the memory—it is a main memory now—smaller but faster intermediate memories are plugged in. These intermediate memories are *caches*, and they hold copies of the main memory's data. Today's architectures typically exhibit two or three caches in a row called cache levels: As the memory access speed depends on its size, the caches are the smaller the nearer they are to the processing unit.
If the processing unit wants to load a record, it does not access the memory directly, but triggers a lookup in the nearest cache. The cache is small and fast compared to the main memory. If the record is contained in the cache, it is taken from the cache. If it is not contained, the cache triggers the next lookup in the next bigger cache or in the main memory, respectively. The first case is a *cache hit*, the latter case is a *cache miss*.
Each data transfer comes along with a certain overhead due to latency and data consistency management. Thus, caches do not exchange single bits and bytes, but they transfer whole blocks of memory. These blocks are *cache lines*. If a record is not held in a cache, the cache requests the whole cache line comprising the record. As the cache size is small, caches frequently run out of space. In this case, the cache has to swap records back to the bigger caches or the main memory. An algorithm cast in the hardware decides which cache lines are swapped. The sooner a swapped cache line is to be reloaded, the worse the algorithm's decision (*capacity miss*). The decision which line to replace could be optimal, if the algorithm knew an application's upcoming memory access pattern. As the application's access pattern is—without loss of generality—not known a priori to the hardware, a cache swapping strategy is never optimal. Nevertheless, an application should diminish cache line replacements.

cache-oblivious algorithms. Cache aware algorithms exploit knowledge about underlying cache hierarchies and cache properties. Typical cache-aware algorithms block memory accesses, reorder data accesses, and fuse loops of different program parts ([22, 48], e.g.). Hereby, they are parametrised with the cache's properties. Cache oblivious algorithms are algorithms relying on the existence of caches but not exploiting the caches' properties quantitatively.

To measure the cache efficiency, the number of cache accesses and the *cache hit rate* are suitable metrics: The former is a counter, the latter divides the number of cache hits by the number of cache accesses. Because of the restricted size, the cache line policy and the replacement challenges, two data access characteristics[6] determine the cache efficiency. The higher the *spatial locality*, the better the cache hit rate. The higher the *temporal locality*, the better the cache hit rate.

A data access pattern exposes *spatial locality*, if

- the distance between two records a and b in the main memory is small
- whenever the algorithm accesses a and b in a row.

The probability then is high that both records are contained in the same cache line. As a result, there is also no need to swap cache lines back to a bigger cache or the memory because of the restricted capacity.

A data access pattern exposes *temporal locality*, if the time in-between the two accesses to a record a is small. The probability then is high that the cache line holding a still resides in the cache and there is no capacity cache miss.

The two terms are introduced in [48]. Yet, they define both terms on Cartesian grids in the context of iterative equation systems solvers. The definition above generalises the idea to abstract memory access patterns.

Corollary 3.6. *The stack-based grid management is cache-oblivious.*

On the one hand, the stack signature permits only memory accesses exhibiting high spatial locality, as the distance in-between two memory accesses is at most one. On the other hand, the fixed and small number of stacks yields the temporal locality. If there were a dynamic number of stacks, this would not hold. Furthermore, the curve's Hölder continuity ensures that the size of the temporary stacks does not change by an order of magnitude throughout the traversal: Vertices swapped to the temporary stacks are usually reused before long, as the curve meanders back soon.

Many PDE solvers suffer from a lack of spatial and temporal locality. Two examples: First, algorithms for parabolic PDEs often exhibit low temporal locality. These equations typically lead to a sequence of linear equation systems, i.e. one linear equation system per time step is to be solved. A simulation code has to

[6] In this incomplete presentation, actual hardware properties such as cache line associativity are neglected.

choose time steps that are sufficiently small to resolve all the solution's temporal characteristics. The solution then exhibits a smooth behaviour in time—it does not change rapidly from one time step to another. For a given time step, the solver of the linear equation system uses the solution of the preceding time step as initial solution guess. As both the preceding solution and the current solution do not differ significantly, the solver needs only a small number of iteration steps to end up with a sufficiently accurate solution. For each time step, the whole grid is traversed. The algorithm exhibits a low number of computations but big data movements.

Second, multiscale algorithms often exhibit low spatial locality. Typically, developers arrange records belonging to one grid level continuous in the memory. As the number of smoother operations outnumbers the number of inter-level operations, it is important to optimise the data layout for the smoother's memory access. Inter-level operations access different records from different grid levels, and, thus, they do not access the memory in a continuous manner.

The introductions of [35, 39, 63] motivate the development of the stack-based vertex management with the von-Neumann bottleneck and the problems of algorithms not exploiting the memory hierarchy. I by contrast make the cache discussion follow the algorithm's presentation, as the cache arguing carries the inherent danger that the reader mixes up cache efficiency and application's speed. The cache hit rate is only one ingredient of fast numerical algorithms[7], although it might become a more and more important property with all the multicore architectures coming up. Here, multiple processing units have to share one bus and one cache hierarchy, and this sharing makes the von-Neumann bottleneck more severe. In addition, low memory requirements, the algorithm's ability to handle arbitrary adaptivity without any overhead, and an algorithm that is able to handle problems that exceed the memory available (see subsequent section) are of value of their own. A good cache hit rate then in turn is a nice non-functional algorithmic property. It is thus important to prove that the cache properties of the algorithms of [35, 39, 63] carry over for the alternative stack management, although this does not exempt the developer from studying the implementation's performance in great detail.

3.5 Some Realisation Details

On the subsequent pages, implementation consequences arising from the usage of an object-oriented language are discussed within the context of the grid storage and traversal concept. Furthermore, I combine a file swapping strategy with the stack approach and add tree cuts splitting up the k-spacetree horizontally into an

[7] In fact, some cache "optimisations" might even slow down the implementation: Replacing a formula alike $-a_1 + 2 \cdot a_2 - a_3$ with $-a_1 + a_2 + a_2 - a_3$ might improve the cache hit rate but slow down the overall computation due to the rise of operations.

active upper part and a lower part not traversed anymore. The latter feature is especially important for multilevel solvers not traversing the whole spacetree all the time. While the discussion exploits the stack and spacetree idea, it does not add new features or algorithms, but bridges the gap from the pure formalism to a framework.

Peano's implementation is a pure C++ code. Straightforward object-oriented implementations typically exhibit two shortcomings: Their runtime efficiency suffers from an instantiation overhead per object creation, and their objects demand for lots of memory compared to highly optimised code written in a lower level language. Both drawbacks gain weight, if the code models the grid's entities, i.e. vertices and geometric elements, as classes. Yet, I follow the academic object-oriented paradigm rigorously.

The Peano code tackles the construction overhead challenge as it exploits the fly-weight pattern [27] intensely. Hereby, a small, fixed number of instances of vertices, e.g., is created. The object's state then is replaced on-the-fly with data from the stacks, i.e. instead of throwing away old objects and creating new ones, the instance hull is used to handle multiple records.

The Peano code tackles the memory overhead challenge as it uses the precompiler DaStGen [13, 14]. This precompiler transforms annotated C++ classes into memory optimised C++ classes. Sets of booleans, enumerations, and bounded integers are packed into a small number of primitives. A boolean attribute, for example, then requires only one bit instead of a whole byte. Furthermore, Peano distinguishes between persistent attributes and attributes which can be discarded when they are written to the output streams: Before data is written to the output streams, the objects are transferred into an alternative representation holding only attributes required in the next traversal. The precompiler automatically generates the transformation code.

Example 3.5. *The storage order on the input stacks determines the adjacency information of elements and vertices, as it determines which element-vertex combinations are passed to the events. Consequently, there is no need to have a global enumeration of the grid entities. My visualisation though needs a global vertex enumeration, and I thus add a global number to each vertex. As the grids can change from visualisation snapshot to snapshot, these numbers are not persistent, i.e. they are not stored on the input and output stacks. Instead, I throw them away whenever I store the records, and I regenerate them on-the-fly whenever I need them.*

The algorithm deriving the *access* and *even* flag (Algorithm A.2, A.3, A.4, and A.5) is complicated and comes along with lots of integer operations. The Peano code hence derives both properties once when the grid is constructed. Afterwards, it stores the data in the geometric elements.

There are three different stack implementations for the grid management. Two simple implementations are based upon C++'s STL (standard template library)

vector type [53, 70] and a plain array. The first is a dynamic data structure growing with the number of entries. The latter one is a fixed data structure, i.e. the user has to specify the maximum stack size a priori. In exchange, no additional operations are required to manage the stack's size. The last implementation is a combination of fixed-sized arrays and a sophisticated hard disk swapping. It enables the application to handle problem exceeding the main memory, and it is studied on the forthcoming pages. Afterwards, this section discusses some technical details of grid traversals not descending into all the leaves of any depth. Many multigrid algorithms work on one fixed maximum level per traversal, and for these algorithms it would be a waste of computing time to traverse the spacetree's elements belonging to a finer level.

3.5.1 File Swapping

The value of a numerical solution of a PDE is interweaved with the accuracy of the numerical discretisation: The smaller the mesh width becomes, the more accurate and reliable the numerical result. Despite all the sophisticated approaches developed to increase the precision (p-adaptivity, improved boundary approximation, extrapolation, etc.), in the end it hence comes down to make the computational grid as fine as possible[8]. The increase in spatial resolution is bounded by the computing time available and the main memory offered by the simulation system—the available main memory per computing node is a limiting factor. Peano circumnavigates this restriction by applying a hard disk swapping strategy for problems exceeding the main memory. The following section outlines this tailored strategy and explains why it does not lead to a performance breakdown.

Whenever hard disk swapping comes into play, it has to be chosen carefully, as the hard disk is slower than the main memory by magnitudes. Otherwise, it unacceptably throttles the overall simulation. Peano's stack concept fits perfectly to swapping. If the hard disk is the place the records are stored, the stacks in the main memory act as cache. The hard disk holds the persistent data, and the main memory makes the access from the processing unit transparent to the application. It is an obvious idea to transfer the insights on cache efficiency to the manual swapping.

Let there be one array, i.e. one buffer, of fixed size in the main memory per stack. Furthermore, let there be one swap file per stack. As the memory is accessed by push and pop operations—they belong to a last recently used paradigm—it is sufficient to hold the upper part of the stack in the memory. The remaining part of the stack will only be needed after the whole upper part is read by pop operations. The algorithm thus stores it on the hard disk. As soon as a maximal fill threshold for the memory

[8] If an application supports p-adaptivity, it might be possible to increase the polynomial degree instead of choosing a finer grid. In this case, this arguing does not hold. Yet, increasing the polynomial degree in practice is restricted by an insufficient smoothness of the underlying solution. In this case, the user has to switch to hp-adaptivity, and the arguing is justified again.

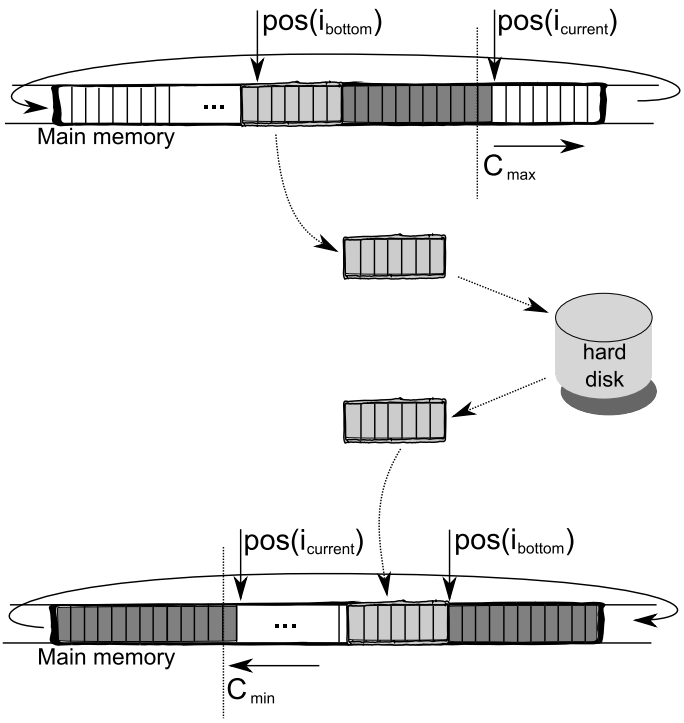


Figure 3.14: If the buffer fill rate exceeds a given maximum (3.8), the part of the stack that was last recently used is swapped to the hard disk (top). If the buffer fill rate falls below a fill threshold (3.8), the most recently used part of the hard disk is reloaded into the main memory's buffer (bottom). Inked entries of the main ring buffer cache data from the disk.

Algorithm 3.5 push operation for a file-based stack implememtation. The predicates and functions are defined in (3.7) and (3.8).

1: **procedure** $push(x)$
2: write to buffer position $pos(i_{\text{current}})$
3: **if** $\mathcal{P}_{\mathcal{M}_{\max}} \wedge i_{\text{current}} - i_{\text{bottom}} > C_{\text{file stack}}$ **then**
4: write $N_{\text{blocksize}}$ buffer entries at $pos(i_{\text{bottom}})$ to swap file
5: $i_{\text{bottom}} \leftarrow i_{\text{bottom}} + N_{\text{blocksize}}$
6: **end if**
7: $i_{\text{current}} \leftarrow i_{\text{current}} + 1$
8: **end procedure**

Algorithm 3.6 pop operation for a file-based stack implememtation. The predicates and functions are defined in (3.7) and (3.8).

```
1: procedure pop
2:     if P_{M_min} ∧ i_current − i_bottom < C_file stack then
3:         load N_blocksize entries from the swap file to location pos(i_bottom) − N_blocksize
4:         i_bottom ← i_bottom − N_blocksize
5:     end if
6:     i_current ← i_current − 1
7:     return record at position pos(i_current)
8: end procedure
```

buffer is passed, the oldest entries within the array are swapped to the associated file. As soon as a minimal fill threshold is under–run, the top swap file partition is loaded into the buffer (Figure 3.14).

The buffer's implementation equals a ring buffer concept. Thus, the main memory buffer for one stack is an array defined by a number N_{blocks} of blocks with $N_{\text{blocksize}}$ entries per block. Blocks mirror the concept of cache lines. Two indices i_{current} and i_{bottom} identify the top element of the stack and the smallest stack element that is still stored in the main memory buffer and not swapped to the file. Since the buffer is a ring buffer, the actual position of an entry i of the stack within the main memory array is given by

$$pos(i) \quad = \quad i \mod (N_{\text{blocks}} \cdot N_{\text{blocksize}}). \tag{3.7}$$

There are two constants $C_{\min}$ and $C_{\max}$ representing a maximal and minimal fill rate threshold within a block. One ends up with the stack access Algorithms 3.5 and 3.6 based upon the predicates

$$\begin{aligned} \mathcal{P}_{\mathcal{M}_{\max}} &= (i_{\text{current}} \mod N_{\text{blocksize}}) \leq C_{\max} \wedge \\ &\quad ((i_{\text{current}} + 1) \mod N_{\text{blocksize}}) > C_{\max} \qquad \text{and} \\ \mathcal{P}_{\mathcal{M}_{\min}} &= (i_{\text{current}} \mod N_{\text{blocksize}}) \geq C_{\min} \wedge \\ &\quad ((i_{\text{current}} - 1) \mod N_{\text{blocksize}}) < C_{\min}. \end{aligned} \tag{3.8}$$

Obviously, writing and reading into or from a swap file is independent from the result and the effect of the **push** and **pop** operation. Thus, they might be deployed into an IO–thread of their own.

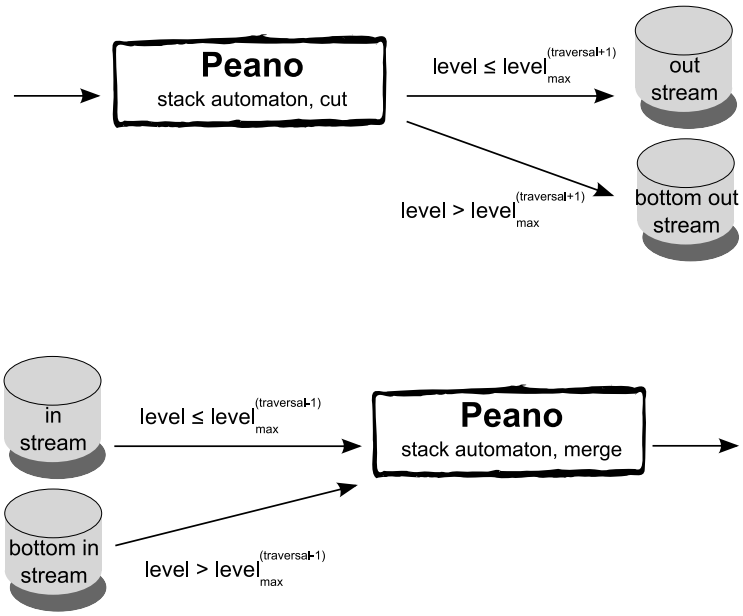


Figure 3.15: A tree cut has to know the maximum traversal level $\ell_{\max}^{(traversal+1)}$ of the subsequent traversal. Records belonging to a higher level are written to the bottom output stream instead of the output stream (top). In the successive iterations, the algorithm does not descend from refined spacetree nodes of level $\ell_{\max}^{(traversal+1)}$, but it treats them as leaves. The counterpart is tree merge.

3.5.2 Horizontal Tree Cuts

All algorithms in this thesis are built on top of a Peano spacetree traversal. Nevertheless, not all of them have to descend into the whole tree all the time: For many algorithms it is sufficient to descend into a given level $\ell_{\max}$. All events for elements and vertices belonging to a level $\ell > \ell_{\max}$ deteriorate to "no operation" (no-op).

Example 3.6. *Consider the multiplicative two-grid algorithm from the upcoming chapter with a $V(\mu)$ cycle. This scheme first traverses the whole grid μ times, and it performs a number of calculations—the smoothing—on each geometric element and each vertex. Afterwards, it ascends, i.e. the consecutive μ traversals perform operations on all elements and vertices belonging to a level smaller than the maximum level. The events for the elements and vertices having maximum level deteriorate.*

It is an obvious idea to make the traversal pass exclusively the part of the tree that is actually needed by the algorithm. Let *traversal* be the actual traversal number. The algorithm knows that the next traversal $traversal + 1$ performs operations for each vertex and element up to a level $\ell_{\max}^{(traversal+1)}$. Whenever a record is written

to the output stream, the algorithm analyses its level. If the level is smaller or equal $\ell_{\max}^{(traversal+1)}$, it is written to the output stream. Otherwise, it is written to the bottom output stream. Throughout traversal $traversal+1$, the algorithm reads from traversal $traversal$'s output stream, and refined elements on level $\ell_{\max}^{(traversal+1)}$ are treated as leaf nodes. The bottom output stream meanwhile is not modified. This process is a *tree cut* (Figure 3.15). The counterpart of the tree cut is a *tree merge*. Throughout the merge, the traversal reintegrates data from the bottom stream.

Obviously, such an approach can work with an arbitrary sequence of cuts and merges, i.e. more than one cut in a row is allowed, as long as the order of the bottom stream fits to the overall stack access scheme.

The tuple $op = (traversal \in \mathbb{N}_0, \ell_{\max}^{(traversal)} \mapsto \ell_{\max}^{(traversal+1)})$ identifies one cut or merge, respectively, throughout traversal $traversal$. $(16, \infty \mapsto 23)$ e.g. denotes that throughout the 16th traversal all records belonging to a level $\ell > 23$ are written to the bottom stack. Before, the whole tree had been traversed. Afterwards, the traversal algorithm descends the spacetree up to level 23, i.e. the maximum level $\ell_{\max}$ changes from ∞ to 23. If the tuple is followed by $(19, 23 \mapsto 15)$, e.g., this denotes that traversal 17 and 18 descend into level 23. Traversal 19 then descends up into level 15 and swaps additional parts of the k-spacetree to the bottom stream.

Let $op_1 op_2 op_3 \ldots$ denote a cut and merge sequence. The sequence has to coincide with the following grammar with start symbol O to make the streams' orders fit to each other:

$$\begin{aligned} O \quad \mapsto \quad & \epsilon \mid \\ & OO \mid \\ & (t, \ell_i \mapsto \ell_j) O (t + 1 + 2 \cdot m, \ell_j \mapsto \ell_i), \\ & \qquad t, m \in \mathbb{N}_0, \quad \ell_i, \ell_j \in \{\infty\} \cup \mathbb{N}_0, \quad \ell_i > \ell_j. \end{aligned} \tag{3.9}$$

There is a merge for each cut reintegrating all the records from to the bottom output stream into the input stream, as the levels ℓ_i and ℓ_j in the two tuples coincide (3.9). The overall cut and merge sequence is ordered in time. As the traversal direction switches after each traversal, and as the bottom output stream acts as bottom input stream throughout the merge, the $2 \cdot m$ in (3.9) ensures that the streams' orders fit to each other. Finally, I postulate for two consecutive transitions $(t, \ell_1 \mapsto \ell_2)(t+1, \ell_3 \mapsto \ell_4)$ that $\ell_2 = \ell_3$ and $\ell_4 < \ell_2 \vee \ell_4 = \ell_1$ hold.

3.6 Experiments

The following experiments analyse the runtime per vertex for a pure grid traversal: the traversal creates the grid and moves data from one stack to the other, while the

Table 3.1: Memory requirements for different dimensions and code constituents. All numbers are given in bytes per grid/traversal entity. The vertices and elements hold only structural data, i.e. they lack PDE-specific properties.

	$d = 2$	$d = 3$	$d = 4$
Traversal automaton	40	56	72
Geometric element	10	14	18
Vertex	2	2	2

traversal events—besides the geometry analysis and the evaluation of refinement criterion—degenerate to no-op. This runtime behaviour is broken down into effects stemming from different ideas presented in this chapter. The experiments were conducted on the Pentium, Opteron and Itanium platform (Appendix B).

A circle, sphere or hypersphere, respectively, is embedded into the unit hypercube for all the experiments, i.e. the geometric setup is the same for all test runs. Besides the grid-related data (Section 3.5), the geometric elements hold an inside/outside flag and the vertices hold an inside/outside/boundary identifier. The program ends up with a very modest memory consumption (Table 3.1), and the memory needs do not depend on the type of architecture, i.e. whether it is a 32 bit or 64 bit system. The latter statement holds if and only if the compiler's memory alignment is switched off.

The traversal automaton encodes—besides level, traversal state variables, and so forth—the spatial position and the current spatial resolution. Both are vectors with floating point entries requiring eight bytes. For each additional dimension, the automaton's size hence increases by two times eight bytes. If an automaton instance is stored in the memory, it is stored on the call stack.

The geometric element encodes both the geometric flag and traversal information such as the *access* and *even* flags. Computing these flags on-the-fly would lead to a significant performance breakdown. The *even* flag is just a d-tuple of boolean flags, the *access* flag consists of $2 \cdot d$ integer values. Hence, the flag's size grows by one integer's size—it is actually realised by a byte—per dimension for the *access* flag, whereas DaStGen [13, 14] encodes all the remaining information within two additional bytes. The realisation does not exploit DaStGen's opportunity to pack integers of bounded size. This could be of value for the *access* flag, and it might reduce the element's size further.

Finally, the vertex holds the geometric information, i.e. whether it is inside, outside or on the computational domain's boundary, and the refinement status. The latter requires six bits, and, in turn, the vertex's state fits into two bytes.

3.6.1 Vertex Access Cost

The first set of experiments compares the runtime per persistent vertex for regular grids to the runtime per persistent vertex for adaptive grids. A sequence of experiments on regular grids with decreasing mesh size was followed by a sequence of experiments exclusively applying the geometric refinement criterion (2.9). Hereby, I chose the minimal mesh size such that the total number of persistent vertices $\mathbb{V}_\mathcal{T} \setminus \mathbb{H}_\mathcal{T}$ is of the same order as the regular grid's vertices. All the experiments were conducted with a stack implementation based upon C++'s STL vector. As the regular grid's number of vertices grows by a factor of 3^d for each additional level, the main memory rigorously restricts the number of possible experiment setups.

The measurements (Figure 3.16) reveal three insights:

1. As hanging vertices are created on-the-fly, traversals on adaptive grids last longer than traversals on regular grids of the same size. This holds for all experiments besides very small, two-dimensional setups with solely a few thousands of grid points. The time spent on the generation of hanging vertices worsens the computing time per vertex ratio. This effect is not observable for $d = 3$ and $\mathbb{V}_\mathcal{T} \setminus \mathbb{H}_\mathcal{T} \approx 1.0 \cdot 10^4$, as the adaptive grid here almost equals the regular one. Therefore, the drawback resulting from hanging vertices occurs only if the adaptivity is sufficiently developed, i.e. if it is a very strong, local adaptivity.

2. The runtime gap between the adaptive and the regular grid for $d = 2$ is invariant of the grid size, i.e. the hanging vertices' overhead does not dominate the overall computing time with a smaller and smaller grid size. This holds although the adaptive grid is extremely adaptive, i.e. it exclusively refines on the domain's boundary. For each refinement flag set, the grid is added at least 5^d additional non-hanging vertices. The raise in computing time spent on these vertices is not outnumbered by the overhead required by the additional hanging vertices.

3. The time per vertex converges to a fixed runtime, i.e. the implementation's performance per vertex is—for sufficiently big problems— invariant of the grid size.

As all experiments were conducted with a dynamic data container, the question is justified whether the application suffers from the dynamic memory management of the STL implementation. To avoid "pollution" effects resulting from the hanging nodes, I reran the experiments for the regular grids with the three different stack implementations described in Section 3.5. The result in Figure 3.17 exhibits two interesting insights.

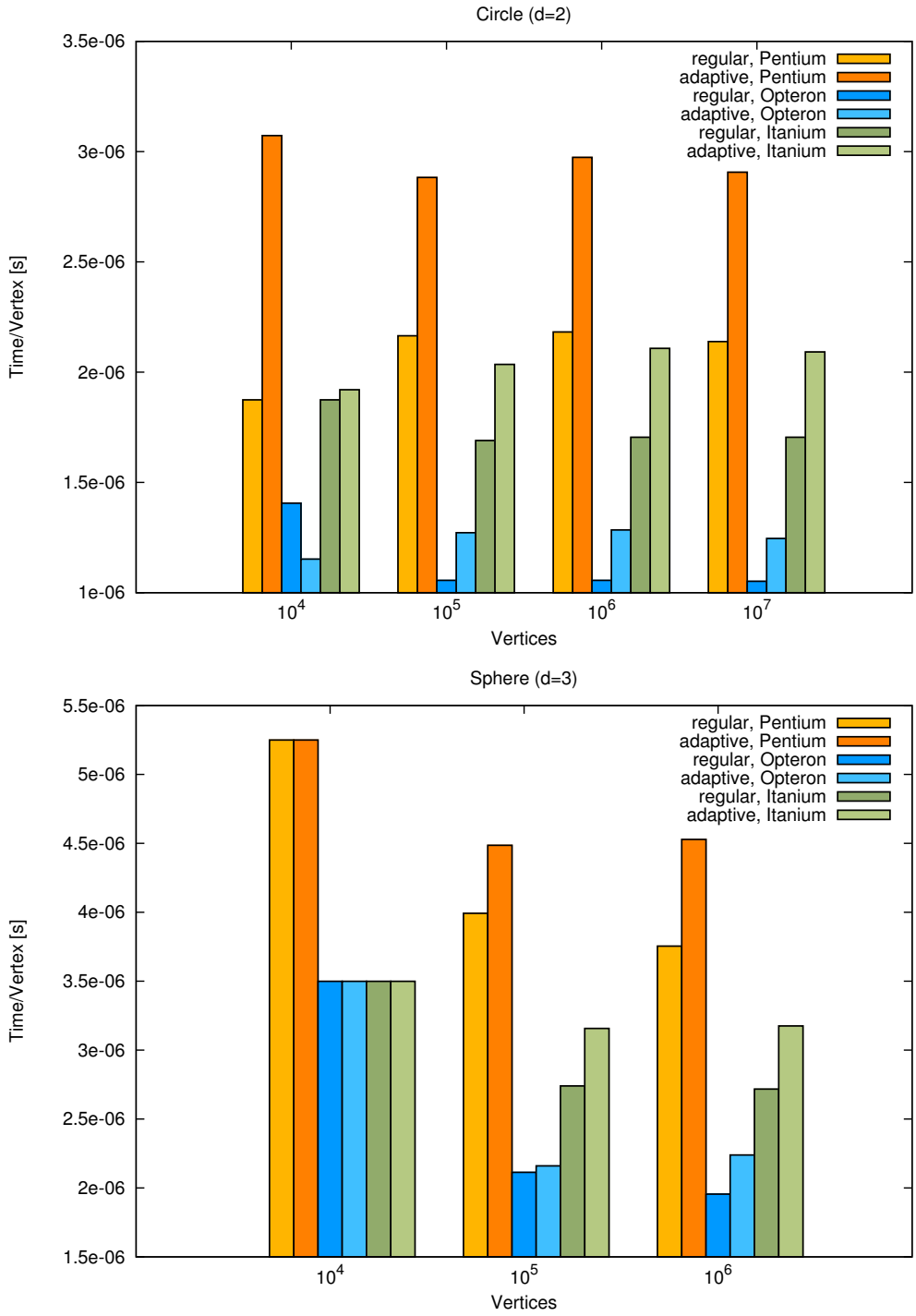


Figure 3.16: A circle (top) or a sphere (down), respectively, are discretised by a k-spacetree with a regular and an adaptive grid.

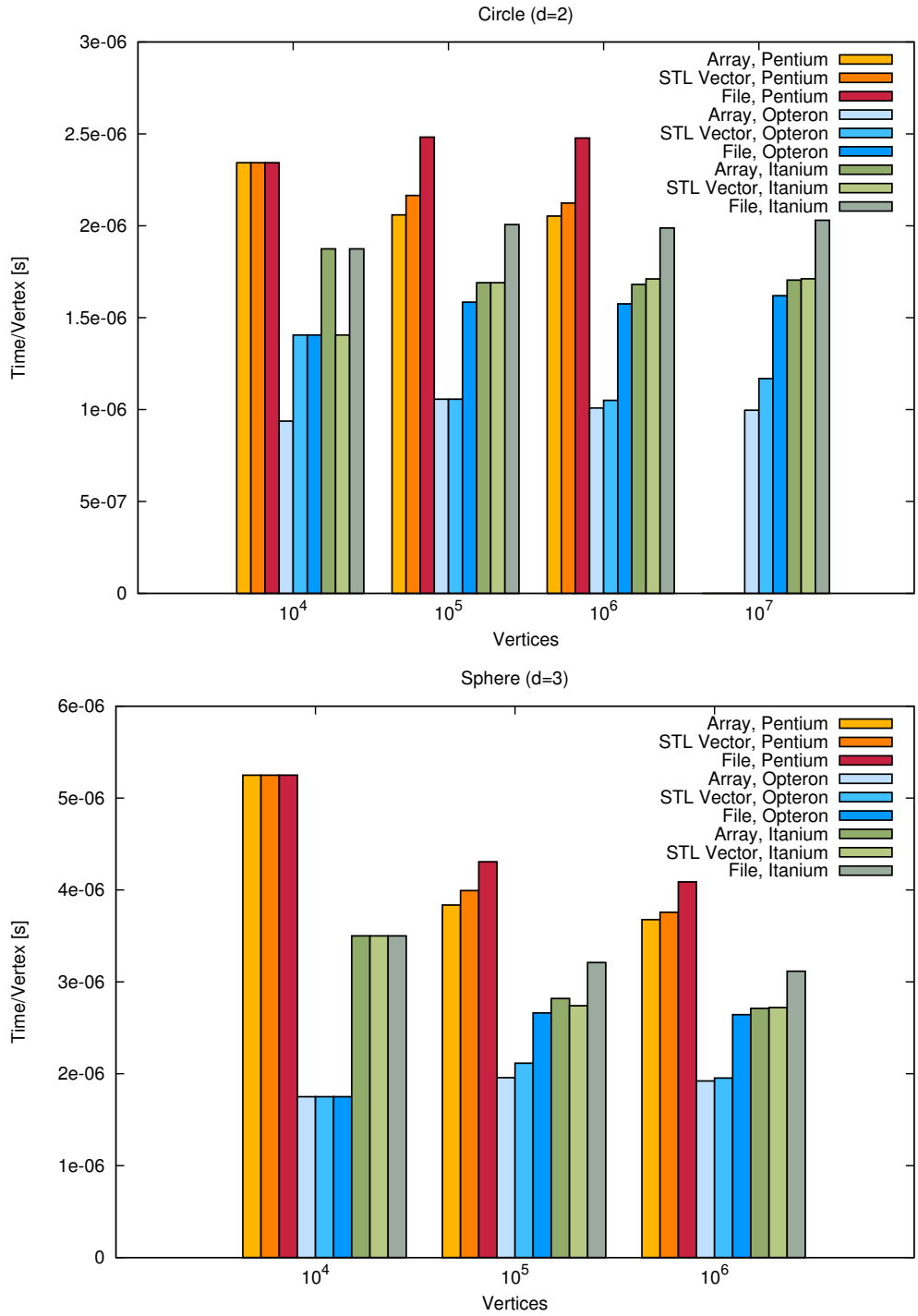


Figure 3.17: A circle (top) or a sphere (down), respectively, are discretised by a regular k-spacetree's grid. Three different stack implementations are compared.

First, the performance penalty induced by the dynamic memory management is negligible. Thus, the two advantages of the STL implementation outnumber the loss in performance: With a dynamic stack memory management, one can compute bigger problems than with a fixed size stack memory. With a fixed size stack memory in turn, one has to know or guess how big the temporary stacks might become. If the guess turns out to be false, the simulation has to be stopped, the stack size guess has to be adopted, and the simulation is to be reran. The STL's vector implementation works on arrays, i.e. if the vector runs out of memory, the implementation allocates a bigger one and transfers the stack's data to it. In return, the implementation also can allocate a smaller amount of memory if the stack's content underruns a given threshold. The maximum experiment size is determined by the available memory in combination with the maximum size of the stacks for an array-based implementation. If the application shrinks stacks on-demand, bigger problems become solvable. For the input and output stacks, the shrinking is straightforward. The maximum size of the temporary stacks is studied in Section 3.6.2.

Second, the usage of the file stacks exhibits a small performance penalty, and this penalty is independent of the problem size. It is an obvious idea to study the effect of the file stacks for even bigger problem. In Section 3.6.4, additional experiments reveal that the algorithm can indeed handle problems significantly exceeding the main memory available with constant cost per degree of freedom. In this experiments, no multithreading was used, i.e. the file stack overhead merges into the constant total cost per vertex. Peano's standard configuration deploys the file management into a thread of its own. The file swapping then is for free, if a core is spent on the overhead studied above.

3.6.2 Maximum Stack Size

The discussion of an advantageous stack implementation lacks the knowledge how big the temporary stacks actually become. The experiments in Table 3.2, Table 3.3, and Table 3.4 track the temporary stack sizes and compare them to the total number of persistent vertices. All experiments study two geometries, a hypersphere and a hypercube, discretised by a regular and an adaptive grid. The results reveal several insights:

1. As both geometries are symmetric, pairs of stacks corresponding to faces with a normal that is parallel to the same coordinate axis have the same maximum cardinality. Thus, there are only d stacks with the same maximum stack size.

2. The temporary stacks' sizes are smaller than the input and output stacks by magnitudes, as the Peano space-filling curve is a local curve. Here is a colloquial description of the term local: Before traversing from one element to an element far away, the curve prefers to traverse the elements around

Table 3.2: Maximum elements per stack for different resolutions ($d = 2$). The upper experiments study a regular grid, the lower block shows results for an adaptive grid based upon the geometric refinement criterion (2.9).

	in/out	temp 1	temp 2		in/out	temp 1	temp 2
Square	4	0	0	Circle	4	0	0
	68	3	7		68	3	7
	132	3	9		132	3	9
	772	13	35		772	13	35
	6,128	41	115		6,128	41	115
	53,352	123	348		53,352	123	348
	473,428	367	1,059		473,428	367	1,059
Square	4	0	0	Circle	4	0	0
	68	3	7		68	3	7
	248	13	21		132	3	9
	788	41	53		552	8	19
	2,408	13	142		2,052	22	35
	7,268	367	393		6,792	36	53
	21,848	1,097	1,130		21,252	57	78
	65,588	3,285	3,325		64,872	81	115
	196,808	9,847	9,894		195,972	102	156
	590,468	29,531	29,585		589,512	130	210

Table 3.3: Experiment from Table 3.2 with $d = 3$.

	in/out	temp 1	temp 2	temp 3		in/out	temp 1	temp 2	temp 3
Cube	8	0	0	0	Sphere	8	0	0	0
	520	15	31	63		520	15	31	63
	3,264	115	171	262		1,032	15	39	85
	36,032	899	1,249	1,361		14,288	115	299	671
	672,088	7,623	10,351	8,994		319,360	899	2,464	5,874
Cube	8	0	0	0	Sphere	8	0	0	0
	520	15	31	63		520	15	31	63
	3,200	115	171	262		1,032	15	39	85
	25,320	899	1,091	1,361		13,000	91	216	562
	222,400	7,623	8,293	8,994		150,464	392	922	2,365
	1,994,120	67,159	69,279	71,221		1,494,720	1,336	3,081	7,858

Table 3.4: Experiment from Table 3.2 with $d = 4$. The experiments in the first part of the table work on regular grids, the experiments in the second part of the table work on adaptive grids.

	in/out	temp 1	temp 2	temp 3	temp 4
Hypercube	16	0	0	0	0
	4,112	63	127	255	511
	42,528	1,063	1,527	2,215	3,264
Hypersphere	16	0	0	0	0
	4,112	63	127	255	511
	8,208	63	159	343	705
Hypercube	16	0	0	0	0
	4,112	63	127	255	511
	42,272	1,063	1,527	2,215	3,264
	856,592	23,015	26,615	31,433	38,885
Hypersphere	16	0	0	0	0
	4,112	63	127	255	511
	8,208	63	159	343	705
	248,448	931	2,407	5,703	12,021

the local elements. This locality property is a direct result from the Peano curve's Hölder continuity [66], and it is exploited and discussed in greater detail throughout Chapter 5 applying the curve for parallelisation. For the grid management, the locality results in small temporary stacks compared to the input and output stack, and because of the small stacks the data access is very local, too: Long sequences of exclusive writes (and corresponding reads) do not occur. Instead, the temporary stack's size is oscillating around a small value. This abets a good cache hit rate further.

3. For regular grids, the pairs of stacks differ in their maximum size significantly. The definition of a standardised traversal explains this difference: Records stored on stacks corresponding to the faces with a normal along the x_1 axis are more likely to be read in the next geometric elements than records stored on other stacks. The standardisation prioritises the temporary stacks.

4. The latter observation and arguing do not hold for strongly adaptive grids.

3.6.3 Cache Hit Rate

Section 3.4 predicts a good cache access behaviour for the Peano algorithm. The figures in Table 3.5 give evidence for this statement. All the figures result from the hardware counters of the Itanium. These counters track the system's behaviour and do not break the measurements down into individual processes.

Table 3.5: Memory access characteristics for different geometries. The data results from the Itanium's hardware counters.

		d	$\frac{\text{L2 Misses}}{\text{L2 References}}$	$\frac{\text{L3 References}}{\text{L2 Misses}}$	$\frac{\text{L3 Misses}}{\text{L3 References}}$	Bus Load
Square	regular	2	0.03722%	1.610	43.08%	$\approx 0\%$
	adaptive		0.05370%	1.776	39.47%	$\approx 0\%$
Circle	regular		0.03243%	1.567	40.35%	$\approx 0\%$
	adaptive		0.04029%	1.378	42.75%	$\approx 0\%$
Cube	regular	3	0.06213%	1.770	30.93%	$\approx 0\%$
	adaptive		0.06778%	1.364	43.77%	$\approx 0\%$
Sphere	regular		0.06908%	1.486	22.63%	$\approx 0\%$
	adaptive		0.05258%	1.346	38.36%	$\approx 0\%$
Hypercube	regular	4	0.07454%	1.493	14.13%	$\approx 0\%$
	adaptive		0.09130%	1.238	30.31%	14%
Hypersphere	regular		0.05671%	1.391	13.83%	$\approx 0\%$
	adaptive		0.12047%	0.880	31.03%	13%

Peano's cache miss rate is negligible (first column). The second column contrasts the number of these misses with the L3 cache accesses, and shows that both figures are in the same order. Nevertheless, the number of L3 cache references exceeds the number of L2 misses. Since the total number of L2 misses is that small, and since a cache simulation predicts a L3 cache miss rate close to zero, the additional accesses and the high L3 cache miss rate are to be caused by alternative (daemons) and operating system processes causing context switches. If the total number of L2 misses were not that small, these switches would not affect the measurements.

All the experiments run on grids with a comparable number of persistent vertices. For extremely adaptive grids, the ratio of the persistent vertices to the maximum size of temporary stacks becomes the bigger the higher the dimension. Thus, the tests with $d = 4$ on adaptive grids lead to long input and output streams, but comparably small temporary stacks[9]. As the input and output stacks do not fit completely into the main memory, each record has to be transferred into the caches by the bus. And as the half-value period in this case is rather small, the bus load becomes measurable. For smaller dimensions, the bus load is negligible, i.e. it was not measured by the hardware counters.

Memory bandwidth is a crucial factor for almost all PDE solvers, and a bandwidth aware algorithm design gains weight with all the multicores sharing one memory

[9]This insight is represented by the figures in Table 3.4 tracking the maximum stack size over the whole traversal time. The underlying principles are discussed in the spacetree chapter and a huge part of the Outlook 2.10 highlights drawbacks, consequences, and improvements concerning the adaptive boundary approximation.

connections coming up. More and bigger caches in combination with cache access optimisations are considered as solution approach for this problem ([44, 48, 77], e.g.). Peano's cache-obliviousness makes the code swim against the stream: In no experiment, the performance was restricted by the memory. The framework thus it particularly promising for applications suffering extraordinary from bandwidth restrictions. Computational fluid dynamics and their flux computations for example are such a field of applications, since their algorithms have to traverse huge grids per iteration while the number of operations per record is typically small [34].

3.6.4 File Swapping Overhead

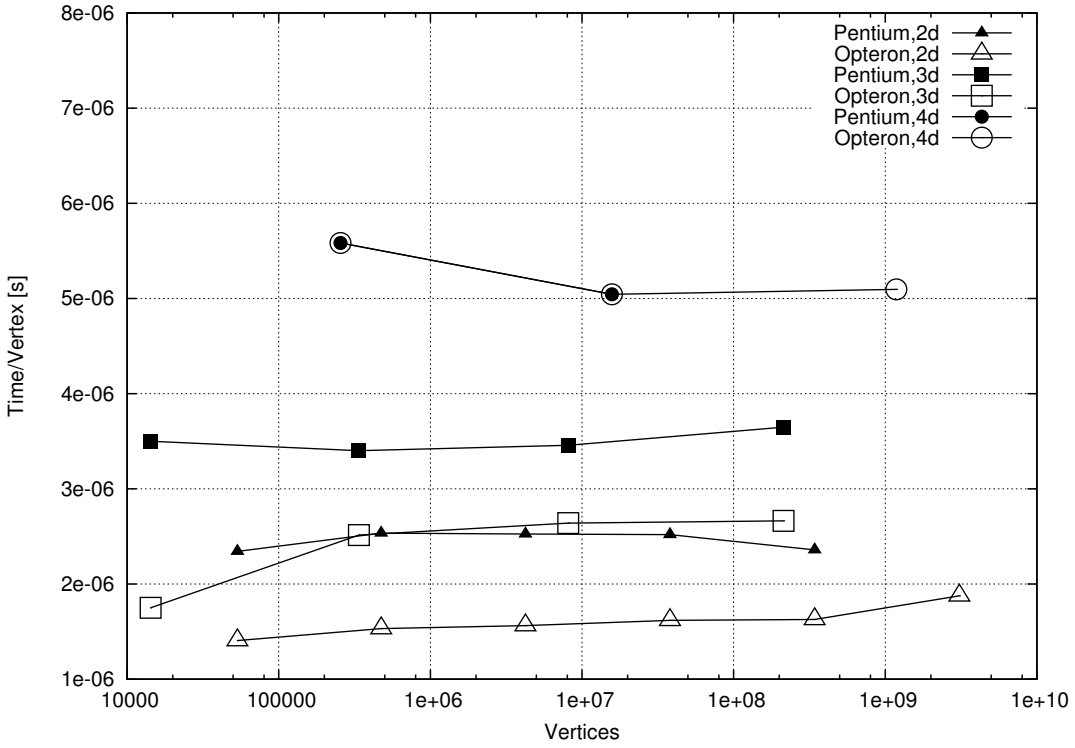


Figure 3.18: Runtime per vertex on two different architectures with the file stacks. The underlying simulation discretises a hypersphere with a regular grid.

Due to the file swapping, Peano is able to solve problems whose datasets do not fit into the main memory. In Section 3.6.1, the runtime overhead resulting from the file swapping is quantified. This overhead is independent of the problem size, i.e. the runtime per vertex is invariant of the grid's mesh width—even for problems not fitting into the main memory anymore (Figure 3.18)[10].

[10]For the Pentium architecture, bigger problems than presented in the figure exceed the address

The file swapping algorithm comprises several constants ($\mathcal{P}_{\mathcal{M}_{\max}}$, $N_{\text{blocksize}}$, and so forth). These constants so far are magic numbers, and they have to be adopted to the concrete architecture (cache hierarchy and properties, available hard disks, connection to the disks, etc.) and application domain. As this thesis establishes algorithms and ideas, a parameter study and optimisation are beyond scope.

Peano's grid management exhibits impressing low memory demands and outstanding high cache hit rates for adaptive grids. The numerical experiments at the end of the thesis nevertheless reveal that the framework suffers from a poor MFlop rate. This is due to the lack of optimisation spent on the current implementation. While the cache-obliviousness does not automatically induce a high performance, the studies nevertheless yield two advantages: On the one hand, an optimisation does not have to care about the cache behaviour—it is already good. On the other hand, the low memory bandwidth requirements enable any optimisation to blow up the individual records stored on within the grid constituents. The latter fact opens the door to a vast field of possible source code optimisations such as precomputing/dictionary techniques and approaches yielding better results per vertex due to an increased number of bytes spent per vertex.

3.7 Outlook

This chapter's sections establish a grid traversal algorithm for $(k = 3)$-spacetrees based on stacks. Both the numerical algorithms and the parallelisation in this thesis are built on top of this traversal algorithm. The whole grid management, traversal, and storage come along with very modest memory requirements, and, since the algorithm requires for a fixed number of $2d + 2$ stacks (two additional stacks are required if tree cuts are switched on), it exhibits high cache hit rates. Furthermore, the runtime per vertex ratio does not grow with an increasing mesh resolution. This holds even if the grid's memory demands exceed the available main memory.

The good memory behaviour of Peano-type algorithms is already studied in [35, 39, 63]. The fundamental new contribution of this chapter thus is the reduction from an exponential number of stacks in d to a linear number of stacks. In the preceding Chapter 2, I establish a flexible definition of k-spacetrees. The chapter here specialises on $(k = 3)$ and derives a grid management for this special case of k-spacetrees. The restriction to $(k = 3)$ results from the genealogy of the algorithm. Nevertheless, it is too restrictive: To make the stack idea work, the algorithm relies on the palindrome property, the projection property, and the invert traversal property. As these properties hold for any k-spacetree with $k \in \{2i + 3 : i \in \mathbb{N}_0\}$, the stack approach works for them, too (Figure 3.19)—an observation already available

space: With $2^{32} \approx 42.95 \cdot 10^8$ and the maximum problem size $4.78 \cdot 10^8$ for $d = 2$, it is obviously that the next bigger problem exceeds the 32 bit threshold.

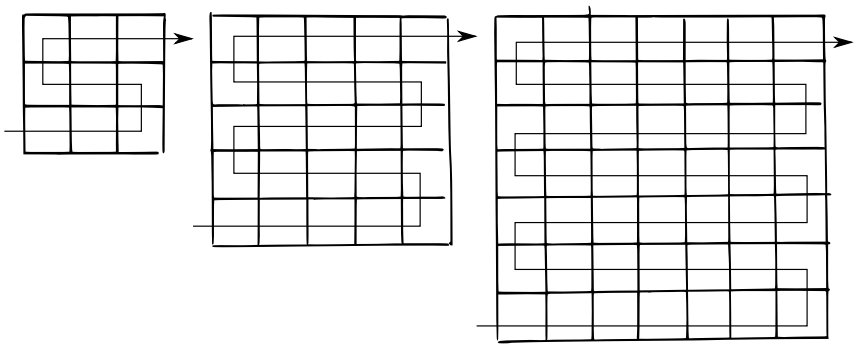


Figure 3.19: The correctness of the stack access scheme relies on the palindrome, the projection and the invert traversal property. All these properties hold for any meander-type traversal based upon an odd number of cuts along each coordinate axis, i.e. for k-spacetrees with $k \in \{2i+3 : i \in \mathbb{N}_0\}$.

throughout the metric discussion of space-filling curves [28]. They call these iterates *rasterised space-filling curves*. The resulting k^d-patches facilitate solver optimisations, as the PDE solvers can process whole k^d-patches. Processing k^d-patches in turn leads straightforward to a couple of further improvements ([23], e.g.):

1. They are traversed lexicographically, which reduces the integer arithmetics resulting from all the mirroring and traversal meanders.

2. The underlying loops are unrolled.

3. As PDE solvers typically need several loops over a grid, these loops are fused.

4. The workload can be deployed to several threads.

To perform the same number of cuts along each coordinate axis for all geometric elements does not fit to every application (anisotropic partial differential equations benefit from a more flexible partitioning, flows dominated by convection benefit from grid aligned with the flow direction, etc.). Hence, it makes sense to decide per axis whether to refine or not. This idea is picked up in Section 2.10, and it fits to the spacetree principle. The increase in flexibility does not affect the underlying idea of a stack-based grid management, if the resulting traversal meanders through the computational domain (Figure 3.20) and, thus, preserves the left/right classification of the vertices—as long as the number of cuts equals an odd number.

If an algorithm has to traverse the spacetree up to a given level, the tree cut technique significantly speeds up the implementation (the effect is studied en detail in the upcoming chapter). Tree cuts in Section 3.5 are parametrised by a global maximum level, i.e. all nodes beyond this level are removed from the traversal tree. This

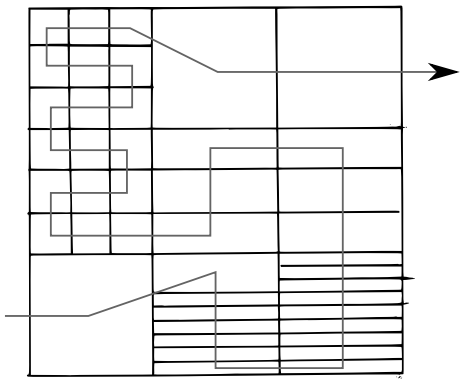


Figure 3.20: Anisotropic ($k = 3$)-spacetrees fit to the stack-based grid management, as long as the traversal remains a continuous meander curve separating vertices into left and right.

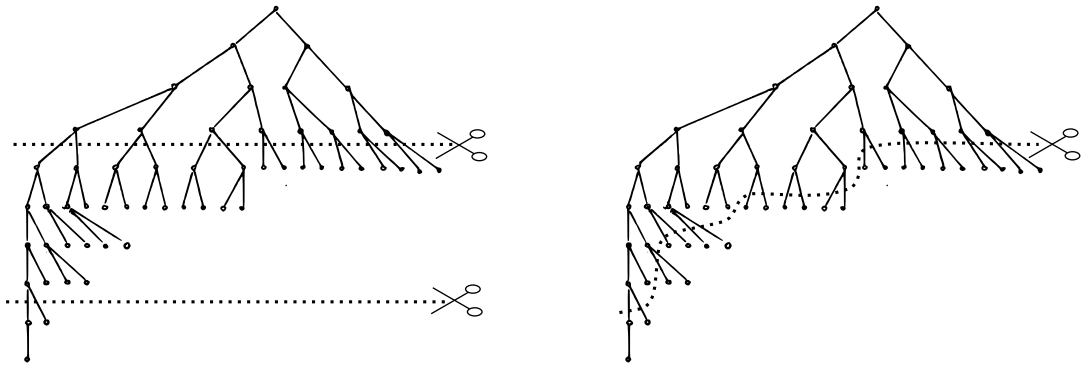


Figure 3.21: A tree cut removes all leaves beyond a given level from the traversal tree, i.e. it "cuts" through a given level horizontally (left). A local maximum traversal level allows individually to select subtrees to be cut from the tree (right).

behaviour motivates the term "horizontal". Nevertheless, some approaches (multigrid algorithms resolving singularities, e.g.) do not yield a global maximum level, but define a maximum level individually for each subtree (Figure 3.21). To remove all the leaves from a tree is an example. Such an extension of the cut mechanism is straightforward: The global variable $\ell_{\max}$ is removed, and each geometric element is added a $\ell_{\max}$ variable of its own. The traversal then decides individually for each subtree, if the records are to be stored on the output or the bottom output stream. As a result, a global stack holding the preceding cut history is inapplicable anymore. It has to be replaced by a more sophisticated control mechanism preserving the tree consistency.

This chapter establishes all the ingredients for a $(k = 3)$-spacetree traversal implementation. In the subsequent chapter, I use this implementation to create a geometric multigrid solver on top of it.

4 Full Approximation Scheme

This chapter realises a finite element solver with a matrix-free, geometric multigrid [73, 10] and d-linear shape functions on adaptive Cartesian grids for the Poisson problem

$$\begin{aligned} -\Delta u &= f, \qquad u, f : \Omega \mapsto \mathbb{R}, \text{ and} \\ u|_{\partial\Omega} &= g, \qquad g : \partial\Omega \mapsto \mathbb{R}, \end{aligned} \tag{4.1}$$

with a sufficiently smooth right-hand side f and a sufficiently smooth Dirichlet boundary condition g. The solver exploits k-spacetrees, and it is embedded into a Peano traversal. It thus demonstrates that the Peano framework allows the implementation of state-of-the-art PDE solvers. In return, the state-of-the-art solver inherits all the nice properties of the framework such as support of dynamic adaptivity, low memory requirements, good cache behaviour, and so forth.

The intention is not to solve a complicated PDE. Instead, I concentrate on algorithmic and numerical principles and choose a simple PDE. Nevertheless, more complicated problems—modified Poisson problems [51, 76], the Navier-Stokes equations and the continuity equation [59, 60], fluid-structure interaction [9], as examples—can be tackled starting from these principles.

The solver is a geometric, multiplicative multigrid implementing a full approximation storage scheme with d-linear shape functions. Geometric multigrid solvers need a sequence of grids giving a refinement cascade. They start with a fine grid, and the succeeding grids then become coarser and coarser. k-spacetrees yield such a hierarchy by construction.

A d-linear shape function on a hypercube is a hat function, i.e. on a Cartesian grid one hat is located at each vertex, and its support covers the 2^d adjacent hypercubes. Such a hat's value equals 1 at the associated vertex and disappears at all other vertices. A linear combination of the different hats approximates the solution with a piecewise d-linear function. As each level of a k-spacetree yields (disconnected) Cartesian grids, there is such an approximation on each k-spacetree level. The overall system is not a hierarchical basis, but a hierarchical generating system [33]. This eases the implementation of a full approximation storage scheme.

Discussing a solver of the linear equation system given on the k-spacetree, [21, 35, 36, 39, 49, 63] study a Jacobi solver preconditioned with an additive multigrid. A conjugate gradient solver with a BPX-type preconditioner for a data structure

similar to a k-spacetree data structure is subject of [2, 69]. In contrast, this chapter establishes a multiplicative approach. The algorithm follows [30] with the realisation embedding the algorithm into the element-wise traversal. For this, I combine [30] with three additional aspects: First, hierarchical generating systems [33] replace the hierarchical basis, as hierarchical generating systems fit better to the k-spacetrees. Second, the bi-partitioning is generalised to k-partitioning. Third, the approach is extended from regular to adaptive grids.

The chapter is organised as follows: An introduction in Section 4.1 establishes the finite element method's weak formulation for d-dimensional hierarchical generating systems or a nodal basis on k-spacetrees. In Section 4.2, the nodal basis leads to a discretisation and stencils for the PDE. The description of standard multigrid ingredients follows. It leads over to the multigrid algorithm tailored to k-spacetrees, and Section 4.4 breaks down the algorithm to the different traversal events. The vertex attributes' lifecycle is analysed afterwards, and I implement a linear error estimator giving a simple, dynamic refinement criterion. A discussion of the global convergence measurement leads over to some experiments in Section 4.6. An outlook closes the chapter.

4.1 Hierarchical Generating Systems

The finite element method computes the solution of (4.1) in the weak formulation

$$\int_\Omega (\nabla u, \nabla \varphi)\, dx = \int_\Omega (f, \varphi)\, dx + \int_{\partial\Omega} (\nabla u, n)\, \varphi dS(x) \qquad \forall \varphi : \Omega \mapsto \mathbb{R}. \tag{4.2}$$

φ is from a set of suitably chosen test functions, whereas n equals the outer normal vector of the computational domain. $(.,.)$ denotes the inner scalar product. The weak formulation (4.2) is accompanied by suitable, weak boundary conditions, and its solution is from a Sobolev space $H^1(\Omega)$ [8].

$H^1(\Omega)$ consists of an infinite number of functions. A finite element method therefore approximates $H^1(\Omega)$ with a finite dimensional function subspace $H^1_{\mathrm{h}}(\Omega_{\mathrm{h}}) \subset H^1(\Omega)$. Given a set of *shape functions* ϕ_i defining a basis of $H^1_{\mathrm{h}}(\Omega_{\mathrm{h}})$ abbreviated as H^1_{h}, the solution of the weak problem within the subspace is a linear combination

$$u \;\mapsto\; u_{\mathrm{h}} = \sum_{i=1}^{|H^1_{\mathrm{h}}|} u_i \phi_i, \text{ i.e.}$$

$$\int_\Omega (\nabla u, \nabla \varphi)\, dx \;\mapsto\; \sum_{i=1}^{|H^1_{\mathrm{h}}|} u_i \int_\Omega (\nabla \phi_i, \nabla \varphi)\, dx \quad \forall \varphi \in H^1\ d, u_i \in \mathbb{R},$$

of the shape functions. The number of unknowns within this linear combination equals the basis' cardinality. To determine the unknowns, it is sufficient to select

a finite number of test functions. This number has to equal the basis' cardinality. *Ritz-Galerkin* methods underlying the forthcoming text equalise the space of the shape functions—the *ansatz space*—and the test space.

$$\int_\Omega (\nabla u, \nabla \varphi)\, dx \quad \rightarrow \quad \sum_{i=1}^{|H^1_\mathrm{h}|} u_i \int_\Omega (\nabla \phi_i, \nabla \phi_j)\, dx \qquad (4.3)$$
$$\forall j \in 1, \ldots, |H^1_\mathrm{h}| \quad \phi_i, \phi_j \in H^1_\mathrm{h}, u_i \in \mathbb{R}.$$

The weak formulation in (4.3) yields $|H^1_\mathrm{h}|$ linear equations to be solved, i.e. a linear equation system

$$A\,(u_1, u_2, \ldots)^T = b \qquad (4.4)$$

describes the solution of the PDE. This system of linear equations is determined by the *stiffness matrix* A with $A_{ji} = \int_\Omega (\nabla \phi_i, \nabla \phi_j)\, dx$. If the shape functions have local support, A exhibits a sparse pattern, since a test function's support then intersects only a few other shape function's supports, and most integrals under the sum in (4.3) vanish. This idea underlies the forthcoming text.

4.1.1 The Finite Element

The finite element method here works with d-dimensional shape functions, i.e. the function φ with

$$\begin{aligned}
\phi : \mathbb{R}^d &\mapsto \mathbb{R}, \\
\phi_{1\mathrm{D}} : \mathbb{R} &\mapsto \mathbb{R}, \\
\phi_{1\mathrm{D}}(t) &= \begin{cases} t & if \quad t \in [0,1] \\ 1-(t-1) & if \quad t \in [1,2] \\ 0 & else \end{cases}, \\
\phi(x) &= \prod_{i=1}^{d} \phi_{1\mathrm{D}}(x_i)
\end{aligned}$$

gives one reference shape function. For each non-hanging vertex v_i, the *hat function* φ is translated such that its maximum value 1 coincides with the vertex position. Afterwards, the translated hat function is scaled such that its support covers the 2^d adjacent geometric elements. The resulting function φ_i hence equals 1 in one vertex v_i and 0 for each other vertex belonging to the same grid level (Figure 4.1). All the vertices of one grid level introduce a piecewise d-linear function space, and the hat functions define a *nodal basis*.

Definition 4.1. *Each non-hanging vertex $v \in \mathbb{V}_\mathcal{T} \setminus \mathbb{H}_\mathcal{T}$ yields one shape function.*

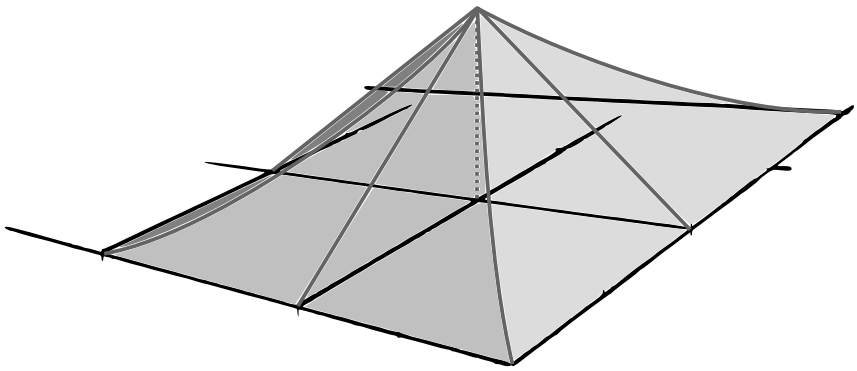


Figure 4.1: A hat function being one in one single vertex. The d-dimensional function vanishes on all other vertices. Its support covers 2^d geometric elements.

The number of shape functions equals the cardinality of the set $\mathbb{V}_\mathcal{T} \setminus \mathbb{H}_\mathcal{T}$. The text sometimes ascribes properties of a vertex directly to the corresponding shape function or vice versa. Consequently, the grid symbols $\Omega_\mathrm{h}, \Omega_{\mathrm{h},\ell}$ and $\Omega_{\mathrm{h},\ell}^{\mathrm{adaptive}}$ then represent the piecewise linear ansatz spaces

$$H^1_\mathrm{h}(\Omega_\mathrm{h}), H^1_\mathrm{h}(\Omega_{\mathrm{h},\ell}), H^1_\mathrm{h}(\Omega_{\mathrm{h},\ell}^{\mathrm{adaptive}}) \subset H^1(\Omega),$$

and the shape functions are denoted by $\varphi_v,\ v \in \mathbb{V}_\mathcal{T} \setminus \mathbb{H}_\mathcal{T}$, i.e. I replace the index by the vertex symbol, as v uniquely defines both level and position. u_v then is the *weight* of vertex v in (4.4).

Definition 4.2. *A* finite element *is a geometric element together with the parts of the 2^d hat functions defined on its vertices and covering the element.*

The construction of the d-linear ansatz space corresponds to a nodal point of view. The definition of the term finite element accentuates an element-wise point of view (Figure 4.2). As hanging vertices do not entail a shape function construction, geometric elements "contain" at most 2^d hats.

4.1.2 Generating Systems on k-spacetrees

A set of functions establishing a basis contains solely linearly independent shape functions. As Definition 4.1 does not distinguish between refined and unrefined vertices, the resulting set of shape functions are linearly dependent. They define a *hierarchical generating system.* Let

$$\Omega_\mathcal{T} = \bigcup_\ell \Omega_{\mathrm{h},\ell}$$

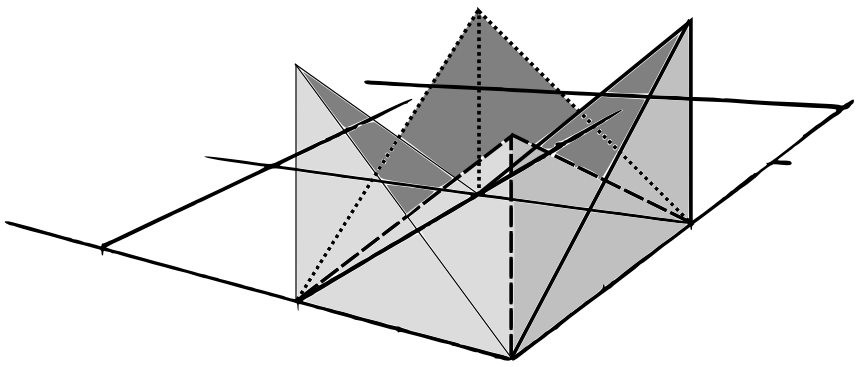


Figure 4.2: A finite element consists of one geometric element and the parts of the grid's hat functions covering the geometric element.

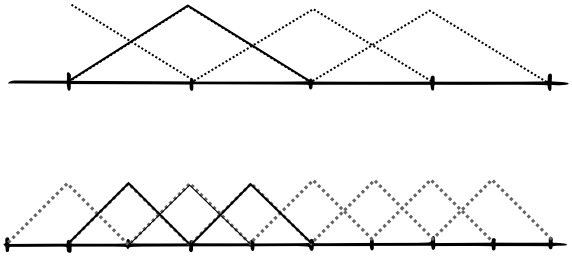


Figure 4.3: Generating system for two levels ($d = 1$). Coarse shape functions on refined vertices can be reconstructed by 3^d fine grid functions (solid fine grid hats). The hats thus give a hierarchical generating system.

denote the union of all the different grid levels. $H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ spanned by the corresponding hat functions—I use $H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ synonym for the generating system, too, i.e. the set also holds the linear dependent hat functions—gives the hierarchical generating system for a k-spacetree.

Example 4.1. *Let $d = 1, k = 2$, and pick out any vertex $\mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$, i.e. a vertex with two adjacent elements. If the vertex is refined, another vertex at the same position on level $\ell + 1$ exists. Add this vertex's hat and its two neighbouring shape functions on $\ell + 1$ with the weights $(\frac{1}{2}, \frac{2}{2}, \frac{1}{2})$. The result equals the shape function on level ℓ (Figure 4.3).*

With the well-defined finite function space, one wants to

- represent functions u_{h} in $H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ and
- apply operators such as the weak Laplacian on these representations. To evaluate an operator, one has to evaluate $u_{\mathrm{h}}(x)$ for any $x \in \Omega$.

These two steps are uncomplicated for a function basis of $H^1_{\mathrm{h}}(\Omega_{\mathrm{h}})$, i.e. a basis for the fine grid system. Let u_{h} represent a function in $H^1_{\mathrm{h}}(\Omega_{\mathrm{h}})$.

$$\forall v = (x, \ell) \in \Omega_{\mathrm{h}} : u_v = u_{\mathrm{h}}(x) \qquad \text{or, the other way round,} \tag{4.5}$$

$$\forall x : \quad u_{\mathrm{h}}(x) = \left(\sum_{v=\in\Omega_{\mathrm{h}}} u_v \phi_v \right)(x). \tag{4.6}$$

Here, the grid points sample the function's course (4.5), and the function in turn is a linear combination of the function space's elements (4.6). The same approach obviously is not suited for a hierarchical generating system (Ω_{h} in (4.5) is replaced by $\mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$) where the functions are not linearly independent, as

- a concatenation of (4.5) and (4.6) does not yield the identity. Both the generating system and the nodal basis have the same range, i.e. each element in $H^1_{\mathrm{h}}(\Omega_{\mathrm{h}})$ can be represented in $H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ and vice versa (Figure 4.3 and Figure 4.4). Nevertheless, a naive reference system transformation followed by an inverse reference system transformation does not result in the original function. Furthermore,

- the scheme does not exploit the additional degree of freedom resulting from the linearly dependent entries of the generating system, i.e. the generating system does not yield an additional benefit.

I apply the following strategy instead: Let $h : H^1(\Omega) \mapsto H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ map a function to a vector of coefficients in the hierarchical generating system. Such an h decomposes any function into a set of hats belonging to different levels. It consequently decomposes a function into its frequencies. h is free to exploit any freedom resulting from the non-uniqueness. In turn, there is a $\hat{h} : H^1_{\mathrm{h}}(\Omega_{\mathcal{T}}) \mapsto H^1_{\mathrm{h}}(\Omega_{\mathrm{h}})$ mapping this representation back to the fine grid. $\hat{h}^{-1}$ does not exist. Nevertheless $\hat{h} \circ h = id, \quad \forall u \in H^1_{\mathrm{h}}(\Omega_{\mathrm{h}})$. In this thesis, I use two pairs of $(h, \hat{h})$ discussed on the following pages. The first representation choice is well-suited for a (Laplacian) operator evaluation, the second for inter-level information transfer. Switching from one representation to another then simplifies the implementation of a multigrid scheme.

Nodal Representation

In a nodal representation, each vertex samples the function to be represented. As shape functions of refined vertices can be removed from the generating system without reducing the system's rank, the projection $\hat{h}$ from $H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ to the nodal repre-

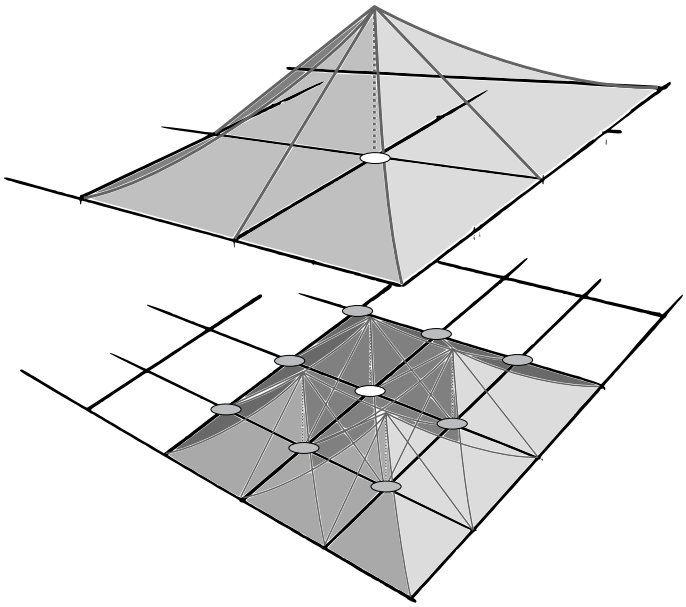


Figure 4.4: Generating system for two levels, $k = 2$, and $d = 2$. Nine shape functions on the fine grid can be combined into one coarse grid shape function. Four of them are illustrated.

sentation does not take them into account (4.8).

$$\begin{aligned}
h : H^1(\Omega) &\mapsto H^1_{\mathrm{h}}(\Omega_{\mathcal{T}}), \qquad \text{with} \\
\forall v = (x,\ell) \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} : \; u_v &\leftarrow u(x) \qquad \text{and} \\
\forall x \notin \Omega : \quad u(x) &:= 0. \qquad (4.7) \\
\hat{h} : H^1_{\mathrm{h}}(\Omega_{\mathcal{T}}) &\mapsto H^1_{\mathrm{h}}(\Omega_{\mathrm{h}}), \qquad \text{with} \\
\forall u_{\mathrm{h}} \in H^1_{\mathrm{h}}(\Omega_{\mathcal{T}}) : \qquad \left(\hat{h}(u_{\mathrm{h}})\right)(x) &= \left(\sum_{v \in V_{\mathrm{temp}}} u_v \phi_v\right)(x) \qquad \text{and} \\
V_{\mathrm{temp}} &= \{v : v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} \wedge \neg \mathcal{P}_{\mathrm{refined}}(v)\}. \quad (4.8)
\end{aligned}$$

In this scheme, the vertices of the hierarchical generating system hold the *nodal value* of the function (4.7), and an algorithm can evaluate a function directly. Furthermore, the scheme represents a function's shape on each level simultaneously, i.e. the hierarchical generating system yields a multiscale representation. The latter ingredient mirrors the idea of full approximation schemes discussed later.

Hierarchical Basis

The construction of a hierarchical basis starts with a nodal basis on a coarse grid. It then adds linearly independent shape functions corresponding to subsequent refine-

ment levels. For Cartesian grids, the refinement scheme is similar to the k-spacetree construction, but new shape functions are only added for vertices at "new" locations: If there is a new vertex $v = (x, \ell)$ and $\exists \hat{v} = (x, \hat{\ell}),\ \hat{\ell} < \ell$, the new vertex does not yield an additional shape function. If the transformation into the hierarchical system mirrors this idea, the inverse transformation just sums up the different levels' contributions, i.e.

$$\begin{aligned} h : H^1(\Omega) &\mapsto H^1_{\mathrm{h}}(\Omega_{\mathcal{T}}), \qquad \text{with} \\ \forall x \notin \Omega : \quad u(x) &:= 0 \qquad \text{and} \end{aligned} \tag{4.9}$$

$$\begin{aligned} \forall v = (x, \ell) \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} : & \\ level(v) = 0 \Rightarrow u_v &\leftarrow u(x), \\ level(v) > 0 \Rightarrow u_v &\leftarrow u(x) - \sum_{v \in V_{\mathrm{temp}}(level(v))} u_v \phi_v(x). \end{aligned} \tag{4.10}$$

Here, $V_{\mathrm{temp}}(k) = \{v = (x, \ell) : v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} \wedge \ell < k\}$, and $\hat{h}$ is given by

$$\begin{aligned} \hat{h} : H^1_{\mathrm{h}}(\Omega_{\mathcal{T}}) &\mapsto H^1_{\mathrm{h}}(\Omega_{\mathrm{h}}) \qquad \text{with} \\ \left(\hat{h}(u_{\mathrm{h}})\right)(x) &= \left(\sum_{v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}} u_v \phi_v \right)(x). \end{aligned} \tag{4.11}$$

In this scheme, the vertices of the hierarchical generating system hold the *hierarchical surplus* (4.10). To point out that a function identifies the hierarchical surplus, i.e. that a d-linear function stems from a linear combination of hats scaled with the linear surplus, I write $\hat{u}_v$ instead of u_v.

4.1.3 Dirichlet Boundary Conditions

Many finite element codes integrate Dirichlet boundary conditions into the ansatz space: they add tailored shape functions approximating the boundary condition. The test space remains unchanged, as the number of shape functions whose weight is to be determined remains unchanged, too. The solution then is the sum of the weighted shape functions of the ansatz space and the hats approximating the boundary values (Figure 4.5).

This thesis follows such an idea, and the boundary vertices realise the Dirichlet boundary condition of the nearest surface point on the continuous computational domain's boundary according to Section 2.7. This convention lacks a handling of the hierarchical system. Here, the actual "boundary" values depend on both the Dirichlet value and the representation scheme chosen.

First, the nodal representation scheme. Unrefined boundary vertices have the value of the nearest surface point of the continuous domain's boundary. According

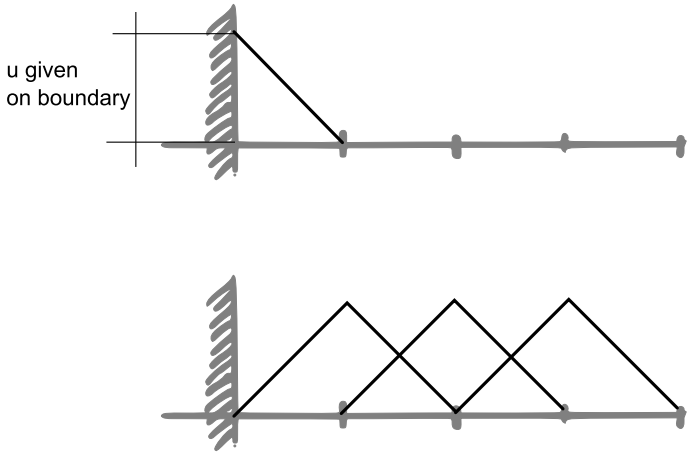


Figure 4.5: Hats at the discretised boundary approximate the Dirichlet boundary, i.e. their weights are defined by the boundary conditions (top). A standard ansatz space is used to solve the problem (bottom). Both function spaces added give a solution of the PDE.

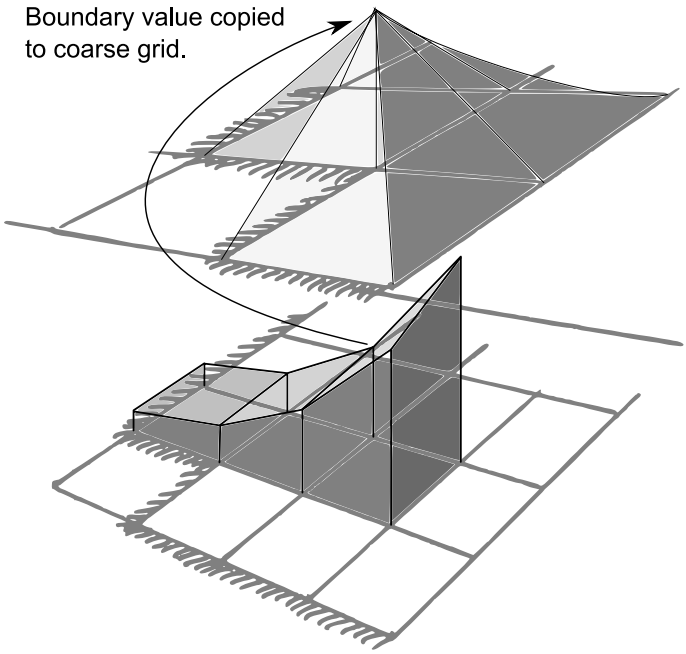


Figure 4.6: In the nodal representation, refined coarse boundary vertices adopts the value of the finer grid, i.e. the coarse boundary values depend on the fine grid approximation.

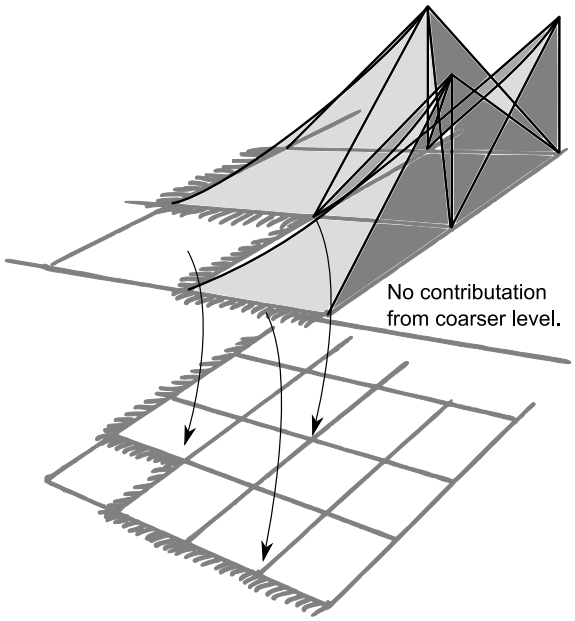


Figure 4.7: In the hierarchical basis, the coarse boundary vertices' values have to vanish, as the sum of coarse and fine grid shape functions may not be discontinuous.

to (4.7), the values of refined boundary vertices equal the finer level's solution at the same place (Figure 4.6).

Let $\wp : \partial\Omega \times \mathbb{V}_{\mathcal{T}} \mapsto \partial\Omega_{\mathrm{h}}$ map the continuous boundary to a discrete vertex position along the shortest distance. In accordance with the nodal representation scheme, the boundary vertices are given by

$$\begin{aligned} \forall v = (x,\ell) \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} : \ \mathcal{P}_{\text{boundary}}(v) \wedge \neg\mathcal{P}_{\text{refined}}(v) : \ u_v &= u(\wp^{-1}(x)), \\ \mathcal{P}_{\text{boundary}}(v) \wedge \mathcal{P}_{\text{refined}}(v) : \ u_v &= u_{(x,\ell+1)}, \end{aligned} \tag{4.12}$$

and the value of a Dirichlet boundary point in a stationary problem can change due to additional refinements at the boundary. When the discrete computational domain grows monotonously towards the exact domain (2.8) due to a refinement, the precision of the boundary approximation on coarser levels in the hierarchical system thus improves at the same time, too (4.12). As a result, it is important to make (coarse) boundary vertices persistent. Otherwise, their value had to be reconstructed analysing all the fine grids' values. Such a reconstruction needs data from finer levels—the values are synthesised [46]—and hence introduces an additional depth-first traversal.

Second, the hierarchical basis. In accordance with h and $\hat{h}$, unrefined boundary vertices have the value of the nearest surface point on the continuous domain's boundary. As there is no other vertex at the same position belonging to a finer

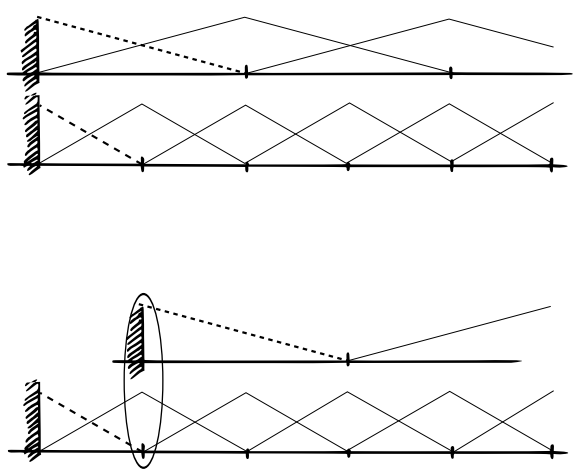


Figure 4.8: Coarse grid and fine grid are summed up. If hats at the boundary exist (dotted hats), the sum is continuous if and only if the coarse and the fine grid boundary coincide.

level, their values determine the discrete domain's boundary (4.10). The values of refined boundary vertices result from the mapping rule (4.11) and from the fact that the nodal representation $u_\mathrm{h} \in C$: As the hat functions at the boundary are tailored to the discrete computational domain (the hat's support vanishes on outer elements), coarser boundary weights have to vanish (Figure 4.7). Otherwise, the d-linear approximation would become discontinuous wherever the fine grid extends the coarse grid's computational domain (Figure 4.8)

For the hierarchical basis, boundary vertices thus have

$$\begin{aligned} \forall v = (x,\ell) \in \mathbb{V}_\mathcal{T} \setminus \mathbb{H}_\mathcal{T} :\ \mathcal{P}_\mathrm{boundary} \wedge \neg\mathcal{P}_\mathrm{refined} :\ u_v &= u(\wp^{-1}(x)),\ \text{and} \\ \mathcal{P}_\mathrm{boundary} \wedge \mathcal{P}_\mathrm{refined} :\ u_v &= 0. \end{aligned}$$

The coarsest grid holds a nodal representation u_v of the solution supplied with homogeneous boundary values. All finer grids' weights denote the hierarchical surplus $\hat{u}_v$. Weights of fine grid shape functions belonging to a coarse grid vertex's position are zero. With all these shape functions removed—they are scaled with zero anyway—the scheme gives a hierarchical basis representation. Again, the value of a Dirichlet boundary vertex can change throughout the computation due to a refinement. Yet, it does not depend on the actual solution on the fine grid.

4.2 Stencils and Operators

The finite element method approximates the solution of a PDE with a finite dimensional function basis due to the equation system (4.4). To set up the involved

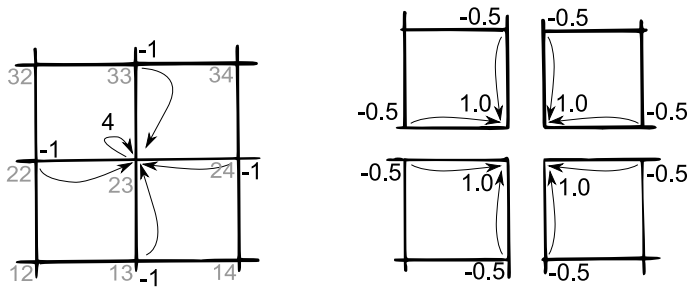


Figure 4.9: The grey numbers give the grid's enumeration. Let for example $\text{tmp}_{23} = 4 \cdot u_{23} - u_{22} - u_{24} - u_{13} - u_{33}$. The stencil (left) illustrates this operator. An element-wise operator evaluation splits up the operator among the cells and sums up the result (right).

stiffness matrix explicitly according to (4.3) is far from trivial, if the k-spacetree's fine grid is not regular. Nevertheless, the definition of the shape functions delivers the complete set of tools required for this task, and the construction process is describable by a set of small matrices. This section defines them. The subsequent text then derives a scheme that solves (4.4) without setting up any global matrix. It is based exclusively on a splitting of the underlying small construction matrices. The thesis' introduction already highlights the advantages of such a matrix-free approach. As a result, the global matrices are used for the description of the equation system solver, but they do never occur in the implementation.

Each row of the stiffness matrix corresponds to one test function ϕ_j in (4.3). The test function is an element of $H^1_{\text{h}}(\Omega_{\text{h}})$, i.e. each test function corresponds to one non-hanging vertex of the fine grid, and its support covers the 2^d adjacent elements. The sum of integrals thus degenerates to a set of 2^d integrals. For all the other integrals in (4.3), the test function vanishes.

Select $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ with v surrounded by non-hanging vertices. The integral that yields the stiffness matrix's line corresponding to test function φ_v then becomes

$$\begin{aligned}
\sum_{i=1}^{|H^1_{\text{h}}|} u_i \int_\Omega (\nabla\varphi_i, \nabla\varphi_v)\, dx \quad &\mapsto \quad \sum_{v_i \in vertex(element(v))} u_{v_i} \int_\Omega (\nabla\varphi_{v_i}, \nabla\varphi_v)\, dx. \\
&= \quad \underbrace{\sum_{v_i \in vertex(element(v))}}_{|vertex(element(v))|=3^d} u_{v_i} \sum_{e \in element(v)} \int_e (\nabla\varphi_{v_i}, \nabla\varphi_v)\, dx,
\end{aligned}$$

involving the weights of 3^d vertices in total. The shape functions φ are d-linear on each geometric element. Hence, there is an analytical expression for the integral, and, given a global enumeration of the vertices, the sum defines the matrix's row

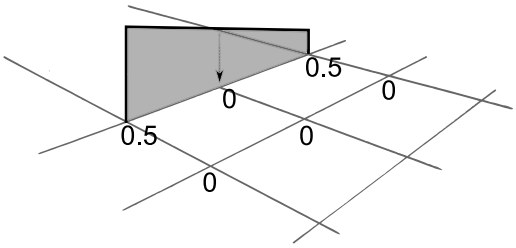


Figure 4.10: The value of a hanging node is interpolated from the coarser grid linearly.

corresponding to test function v. For $d = 2$, this matrix row looks alike

$$\left(\dots -\frac{1}{3} \dots -\frac{1}{3} \dots -\frac{1}{3} \dots -\frac{1}{3} \dots \frac{8}{3} \dots -\frac{1}{3} \dots -\frac{1}{3} \dots -\frac{1}{3} \dots -\frac{1}{3} \right),$$

with ... comprising lots of zero entries. The entries $\frac{-1}{3}$ result from adjacent vertices. $\frac{8}{3}$ scales the weight of the test function's vertex.

While a global vertex enumeration underlies the stiffness matrix, the stencil representation denotes the interplay of the vertices' weights, i.e. the entries of one row of the stiffness matrix, without a vertex enumeration. It exploits the spatial alignment of the unknowns (Figure 4.9). For $d = 2$,

$$\begin{bmatrix} \frac{-1}{3} & \frac{-1}{3} & \frac{-1}{3} \\ \frac{-1}{3} & \frac{8}{3} & \frac{-1}{3} \\ \frac{-1}{3} & \frac{-1}{3} & \frac{-1}{3} \end{bmatrix} = \frac{1}{3} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

defines the stencil for the Laplacian with a d-linear ansatz space on Cartesian grids. Its entries are in $\mathcal{O}(h^{d-2})$, where h is the mesh width of the 2^d adjacent elements. To derive the stencils for further operators is straightforward.

If a vertex neighbouring the test vertex is a hanging vertex, the Laplacian stencil is inapplicable directly, as a hanging vertex does not have a shape function. Instead, shape functions belonging to the coarser grid determine the function's value at the hanging nodes with (4.8) as well as (4.11) yielding the same values. Both formulas define the value at a hanging vertex as a sum of coarser shape functions, i.e. they give an *interpolation* (Figure 4.10). I use the term prolongation as synonym. The interpolation also can be written down as interpolation stencil.

To evaluate the Laplacian for a vertex that neighbours hanging vertices, the interpolation scheme first of all defines the function's values at the hanging vertices. Then, it applies the stencil. Such an evaluation equals a cascade of stencil applications, as the coarse vertices themselves might be hanging vertices again.

Stencils describe the interplay of different vertices. To apply a stencil within a Peano traversal, the algorithm has to have access to all vertices affected by a stencil and to all vertices determining a stencil's image. For the coarsening in (4.7), e.g.,

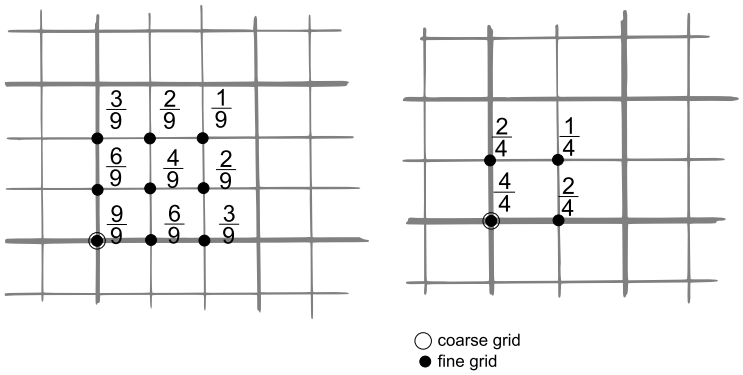


Figure 4.11: Element-wise prolongation for $d = 2$, $k = 3$ (left) and $k = 2$ (right).

these data are available throughout a bottom-up step within the k-spacetree. This is not always the case: The stencil of the Laplacian, e.g., needs all surrounding vertices, but the traversal has access to at most two different geometric elements and their vertices at one time.

In such a case, the algorithm splits up the stencils additively into their element-wise contributions (Figure 4.9 for the Laplacian in $d = 2$). In the finite element world, such a splitting reflects the decomposition of the integral or the hats' supports, respectively, into integrals over individual elements. The result then is to be accumulated within the vertex, i.e. the vertex's result variable is set to zero by the event $touchVertexFirstTime$. Throughout the $enterElement$ or $leaveElement$ events, the result variable is incremented corresponding to the split-up stencil. If the event $touchVertexLastTime$ is triggered, the result variable holds the stencil's image. This algorithmic principle shapes Section 4.4.

4.3 Multigrid Ingredients

The algorithm solves the equation system

$$Au_{\mathrm{h}} = b$$

iteratively without a setup of the stiffness matrix A: At the program startup, all the weights of the inner vertices are zero as initial guess. The algorithm then traverses the grid several times and applies the solver. For this, matrix-vector products are evaluated on an element basis and their results are immediately written back. The matrix-vector product itself is not stored. After several traversals equaling solver iterations, the nodal representation on the generating system is the solution to the linear equation system.

The following pages run through the ingredients of the multigrid solver and identify the operators/matrices involved. They thus provide the third aspect, the miss-

ing link, to realise the solver: With a function representation—in fact two different variants—and the stencils at hand, the algorithm has to know what to do with them, i.e. what variables act as preimage for the stencils and what happens with the result. In this context, the section also reveals which function representation scheme fits best for an algorithmic step.

4.3.1 Jacobi Solver

At the core of the multigrid solver is a Jacobi iteration applying the transition

$$\begin{aligned} u_{\mathrm{h}} &\mapsto u_{\mathrm{h}} + \omega \; diag^{-1}(A) \, (b - A u_{\mathrm{h}}) \\ &=: u_{\mathrm{h}} + \omega \; diag^{-1}(A) \; r, \end{aligned} \tag{4.13}$$

where $\omega \in]0,1[$ is a relaxation factor parameterising the updates. The auxiliary variable r is the *residual*, and $diag(A)$ extracts the diagonal from A. Section 4.4.1 explains why Jacobi is a natural choice within the Peano world.

For a fixed grid level ℓ, an algorithm can apply the update formula (4.13) yielding a Jacobi solver for all non-hanging vertices belonging to level ℓ, as long as all the vertices hold the nodal value of the function (not the hierarchical surplus $\hat{u}_{\mathrm{h},\ell}$). Let thus hold all vertices on level ℓ hold the nodal value. A matrix-free Jacobi for a fixed level ℓ traverses all vertices $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ with $level(v) = \ell$. They hold $u_{\mathrm{h},\ell}$. For each vertex, it computes the value r_v. It depends solely on the neighbours of the non-hanging vertex. These neighbours belong to ℓ, too, and their value is interpolated from the coarser grids if they are hanging. As soon as r is available for all vertices, the algorithm updates all vertices according to (4.13). Hereby, the diagonal element is taken from the stiffness matrix' stencil. The weight of each vertex then represents the new value of $u_{\mathrm{h},\ell}$.

If the k-spacetree yields a regular grid and ℓ equals the height of the spacetree, (4.13) gives a Jacobi solver for the PDE. If the k-spacetree represents an adaptive grid, it does not solve the problem, as regions with a coarser fine grid are not updated at all. The interpolated values resulting from these coarser regions are not updated, too, and the solution on ℓ is not a solution to the PDE but a solution within subregions for artificial boundary conditions resulting from the actual boundary conditions wherever the finest grid covers the computational domain's boundary, and resulting from artificial Dirichlet values given by the hanging vertices.

4.3.2 Standard Multigrid: Smoother and Correction Scheme

Multigrid methods rely on the fact that Jacobi- and Gauß-Seidel-type solvers eliminate errors with high frequency faster than errors with low frequency. They perform a small number γ_1 of solver iterations—they are now called smoothing iterations, as they smooth the error—on the finest grid with level ℓ. If this section refers to regular

grids only, the resulting approximation's quality is now exclusively dominated by low-frequency errors. Then, they derive a *correction equation*

$$\begin{aligned} A_{\ell-1} e_{\mathrm{h},\ell-1} &= R r_{\mathrm{h},\ell} \\ &= R(b - A_\ell u_{\mathrm{h},\ell}) \end{aligned} \tag{4.14}$$

on a level ℓ to approximate the error $e_{\mathrm{h},\ell-1}$ on the next coarser grid. $R(b - A_\ell u_{\mathrm{h},\ell})$ is the coarse grid right-hand side, and A_ℓ equals the stiffness matrix. The coarser grid is determined by the k-spacetree's coarser level, and, thus, this approach is a *geometric multigrid*. The correction equation has to eliminate the remaining error starting with the initial guess $e_{\mathrm{h},\ell-1} = 0$. As the correction problem is defined on a coarser grid, it is smaller and, thus, easier to solve than the original problem. Furthermore, the low frequency errors in the original systems have a higher frequency relative to this grid due to the bigger mesh width. Multigrid methods apply the coarsening idea recursively, i.e. they perform again only a small number of smoothing operations on the correction equation and then apply the coarsening transformation again. Finally, the error correction from (4.14) is transported back to the fine grid, added to the fine grid approximation

$$u_{\mathrm{h},\ell} \mapsto u_{\mathrm{h},\ell} + P e_{\mathrm{h},\ell-1} \tag{4.15}$$

using an operator P, and the algorithm performs another small number of γ_2 smoothing iterations. The overall procedure is a V-cycle (Algorithm 4.1). The letter V results from the grid transitions: The algorithm starts with the finest grid, ascends to the coarsest grid and returns to the fine grid. $V(\gamma_1, \gamma_2)$ gives the exact number of smoothing iterations per level (Figure 4.12). There is a couple of alternative cycles based upon the V-cycle, but these modifications are not discussed as they do not alter the underlying idea and algorithmics.

For the time being, the operators $P, A_{\ell-1}$ and R are to be defined. P transports the correction to a finer level, i.e. it is an interpolation from a coarse grid function to the fine grid. Since the coarse grid in a k-spacetree holds shape functions, it is an obvious idea to use the hat's shape for this interpolation, i.e. the hierarchical geometric structure determines the prolongation (Section 4.2). Additional sub- and superscripts determine the preimage or image, respectively, of the operator.

R is a *restriction* transporting the fine grid residual to the coarser grid. Two natural restriction operators arise for k-spacetrees: A coarsening operator picks out all the fine grid vertices whose position coincides with a coarse vertex. Then, it copies their values to the coarse grid vertices. Such a trivial coarsening operator is implicitly introduced in (4.7). In the following, let $C_\ell^{\ell-1}$ be a coarsening applied on level ℓ and overwriting vertices on level $\ell - 1$.

The alternative restriction results from a weighted summation of the fine grid vertices: Hereby, a coarse vertex's weight is determined by all fine grid values covered

Algorithm 4.1 Blueprint of a correction scheme for regular grids. It is started on the finest mesh level, i.e. the initial ℓ_{active} equals the spacetree's height.

1: **procedure** $cs(\ell_{\text{active}})$

2: Compute an approximation of the solution on the fine grid level ℓ_{active}:

$$A_\ell u_{\text{h},\ell} = b_\ell.$$

A fixed number of Jacobi iterations for this problem is sufficient (presmoothing).

3: Set up the right-hand side of the coarse grid correction (4.14)

$$b_{\ell-1} = R\,(b_\ell - A_\ell u_{\text{h},\ell}) = R\ r.$$

4: Erase variable $e_{\text{h},\ell-1} = u_{\text{h},\ell-1} \leftarrow 0$ approximating the error/correction on the coarser grid.

5: Recursive call: $cs(\ell_{\text{active}} - 1)$

6: Prolongate coarse grid correction back to solution:

$$u_{\text{h},\ell} \mapsto u_{\text{h},\ell} + P e_{\text{h},\ell-1}$$

7: Improve solution on the fine grid level (postsmoothing): ℓ_{active}:

$$A_\ell u_{\text{h},\ell} = b_\ell.$$

8: **end procedure**

by the coarse hat's support. The weight of the individual contributions equals the interpolation stencil P, i.e. the restriction is the transposed prolongation. Prolongation P and restriction R in combination implement a *full weightening*. Again, $R_\ell^{\ell-1}$ denotes the information transport from level ℓ to $\ell-1$, and, thus, $R_\ell^{\ell-1} = \left(P_{\ell-1}^{\ell}\right)^T$. Whenever unambiguous, I omit the source and destination index for the operators.

Finally, the matrix A is to be determined for the individual levels. I follow the *Galerkin multigrid* idea [73, 10] with

$$A_{\ell-1} = R_\ell^{\ell-1} A_\ell P_{\ell-1}^{\ell}. \tag{4.16}$$

Another approach is to derive A_ℓ per level from the weak form and the shape functions. As $H^1_{\mathrm{h}}(\Omega_{\mathcal{T}})$ contains shape functions on each level, this is also plausible. I return to this fact later in the context of adaptive grids.

The Galerkin multigrid idea has at least three valuable properties. First, the correction scheme matrices are well-defined via the restriction and prolongation, i.e. there is no need for a space construction on coarse levels. Second, the prolongation and restriction operators are invariant with respect to scaling. With $R \mapsto c \cdot R, c \in \mathbb{R} \setminus \{0\}$, both the right-hand side of the correction equation (4.14) and the correction operator are scaled by c. For $P \mapsto c \cdot P, c \in \mathbb{R} \setminus \{0\}$, the Jacobi update is scaled by c, i.e. the error summation step (4.15) becomes

$$\begin{aligned} u_{\mathrm{h},\ell} &\mapsto u_{\mathrm{h},\ell} + c \cdot P e_{\mathrm{h},\ell-1} = u_{\mathrm{h},\ell} + c \cdot P \left(A_{\ell-1}^{-1} R r_{\mathrm{h},\ell}\right) \\ &= u_{\mathrm{h},\ell} + c \cdot P \left(c^{-1} \cdot P^{-1} A_\ell^{-1} R^{-1} R r_{\mathrm{h},\ell}\right). \end{aligned}$$

Third, most theorems on convergence analysis rely on (4.16).

4.3.3 Full Approximation Storage

Correction schemes pave the way to asymptotically optimal solvers for elliptic problems. As soon as the problem becomes non-linear, out-of-the-box correction schemes run into problems. This insight historically motivated full approximation storage schemes—an augmented correction scheme additionally holding a multiscale representation of the solution approximation on each level. The following section elaborates this extension as it falls into place for k-spacetrees. In the end, it also proves of great value for adaptive grids. In the context of Peano, the latter fact is much more promising and motivating than the support for non-linear equations.

Full approximation storage schemes hold a representation of the solution on each level. Before the algorithm determines the correction equation for the coarse grid, it thus creates a representation of the current solution on the coarser levels. The injection C is a straightforward realisation of such a coarsening. More sophisticated implementations are discussed for example in [30]. With a full approximation storage, the geometric multigrid algorithm follows the blueprint in Algorithm 4.2.

Algorithm 4.2 Blueprint of full approximation storage scheme for regular grids. It is started on the finest mesh level, i.e. the initial ℓ_{active} equals the spacetree's height.

1: **procedure** $fas(\ell_{\text{active}})$

2: Compute an approximation of the solution on the fine grid level ℓ:

$$A_\ell u_{\mathrm{h},\ell} = b = b_\ell.$$

A fixed number of Jacobi iterations for this problem is sufficient.

3: Transport the fine grid approximation to the next coarser level:

$$u_{\mathrm{h},\ell-1} = C_\ell^{\ell-1} u_{\mathrm{h},\ell}. \tag{4.17}$$

4: The coarse grid correction (4.14) then becomes

$$\begin{aligned} A_{\ell-1}\left(Cu_{\mathrm{h},\ell} + e_{\mathrm{h},\ell}\right) &= Rr_{\mathrm{h},\ell} + A_{\ell-1}Cu_{\mathrm{h},\ell}, \qquad \text{or} \\ A_{\ell-1}u_{\mathrm{h},\ell-1} &= Rr_{\mathrm{h},\ell} + A_{\ell-1}Cu_{\mathrm{h},\ell} =: b_{\ell-1}. \end{aligned}$$

The first equation adds (4.17) on both sides of the coarse grid equation system.

5: Apply solution idea recursively with $Cu_{\mathrm{h},\ell}$ as intial guess for the solution of the coarse grid system: $fas(\ell_{\text{active}} - 1)$

6: Prolongate the difference between the smoothed coarse grid and the original fine grid value $u_{\mathrm{h},\ell-1}$ to the fine grid, sum up fine representation and this value, and continue.

7: **end procedure**

With the coarse grid shape functions inducing P and an unscaled full weightening $R = P^T$, the coarse grid operator obeys both the Galerkin multigrid idea and a direct discretisation. As a result, the traversal holds one hard-coded stiffness operator and applies it directly on the vertices after it is scaled with h^{d-2}. The d results from the integral over Ω, the reduction by two results from the two derivatives in the integrand. There is no need to hold a coarsened A_ℓ on any level.

4.3.4 Hierarchical Transformation Multigrid Method

A multigrid iteration runs through the individual grid levels. Bottom-up first and then reverse. Throughout the steps up, the presmoothing is performed and the correction equations are set up. Throughout the steps down, the postsmoothing is performed and the difference between the finer and coarser levels is transported back to the fine grid levels. To transport the solution difference back implies that the algorithm has to keep book of the original coarse grid values. In [30], an elegant realisation of this bookkeeping is introduced. It uses the hierarchical representation scheme.

Let $\hat{u}_{\mathrm{h},\ell}$ define the *hierarchical surplus* with

$$u_{\mathrm{h},\ell} = P_{\ell-1}^{\ell} C_{\ell}^{\ell-1} u_{\mathrm{h},\ell} + \hat{u}_{\mathrm{h},\ell}.$$

The *hierarchical residual* then is

$$\hat{r}_{\mathrm{h},\ell} = b_\ell - A\hat{u}_{\mathrm{h},\ell}. \tag{4.18}$$

With this definition, the right-hand side of the full approximation storage scheme becomes

$$\begin{aligned} \hat{r}_{\mathrm{h},\ell} &= b_\ell + APCu_{\mathrm{h},\ell} - Au_{\mathrm{h},\ell} \\ &= r_{\mathrm{h},\ell} + APCu_{\mathrm{h},\ell} \\ \Rightarrow R\hat{r}_{\mathrm{h},\ell} &= Rr_{\mathrm{h},\ell} + RAPCu_{\mathrm{h},\ell} \\ &= Rr_{\mathrm{h},\ell} + ACu_{\mathrm{h},\ell} \end{aligned} \tag{4.19}$$

due to the Galerkin multigrid definition. I here omit the operators' source and destination identifiers. The full approximation scheme corresponds to a nodal representation of the approximation within the generating system. The hierarchical transformation multigrid (HTMG) method switches from the nodal representation to a hierarchical basis to compute a hierarchical residual (4.18) instead of the standard residual. This hierarchical residual also simplifies, besides the bookkeeping, the computation of the restriction's preimage within a $V(\gamma_1, \gamma_2)$-cycle (4.19). See Algorithm 4.3. The switch from a nodal system to the hierarchical surplus also is to be performed for the fine grid vertices: the mechanism strictly follows (4.10).

Algorithm 4.3 Blueprint of full approximation storage scheme for regular grids. It is started on the finest mesh level, i.e. the initial ℓ_{active} equals the spacetree's height.

1: **procedure** $htmg(\ell_{\text{active}})$
2: Perform γ_1 Jacobi smoothing steps for $Au_{\text{h},\ell} = b_\ell$ (presmoothing).
3: Coarse the approximation with $u_{\text{h},\ell-1} = Cu_{\text{h},\ell}$.
4: Determine the hierarchical surplus $\hat{u}_{\text{h},\ell} = Pu_{\text{h},\ell-1}$ on level ℓ. The nodal value on this level is not used anymore, so the algorithm stores $\hat{u}_{\text{h},\ell}$ within the variable $u_{\text{h},\ell}$. The algorithm hence switches the representation of grid level ℓ from a nodal to a hierarchical storage scheme.
5: Compute the hierarchical surplus $\hat{r}_{\text{h},\ell} = b_\ell - A\hat{u}_{\text{h},\ell}$. The hierarchical surplus on hanging vertices equals zero. As $u_{\text{h},\ell}$ holds the hierarchical surplus, the residual equation equals the Jacobi update's residual computation. The residual however is not used to update the solution.
6: Transport the hierarchical residual to $\ell - 1$ as new right-hand side on the coarse grid, i.e. $b_{\ell-1} = R\hat{r}_{\text{h},\ell}$ for the refined vertices on level $\ell - 1$. The unrefined vertices' right-hand side on level $\ell - 1$ is not altered.
7: Apply the scheme with the new right-hand side recursively on level $\ell - 1$: $htmg(\ell_{\text{active}} - 1)$
8: Interpolate the new coarse grid value to the fine grid according to $u_{\text{h},\ell} = \hat{u}_{\text{h},\ell} + Pu_{\text{h},\ell-1}$. This transformation switches the representation of grid level ℓ from a hierarchical system back to a nodal system. Because of the updated coarse grid solution, the interpolated value of hanging vertices on level ℓ differs from the initial interpolated value.
9: Perform γ_2 Jacobi iterations for the system $Au_{\text{h},\ell} = b_\ell$ (postsmoothing).
10: **end procedure**

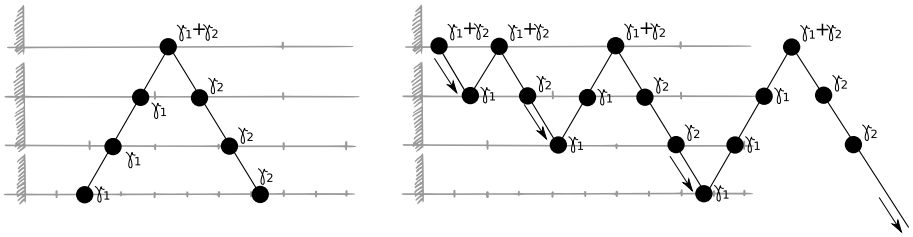


Figure 4.12: V-cycle (left) and F-cycle (right). γ gives the pre- or postsmoothing steps, respectively. The arrow denotes the interpolation with higher order.

4.3.5 Full Multigrid

The accuracy of a numerical approximation is co-determined and, hence, bounded by the discretisation error. In turn, the approximation's error can not underrun the error error induced by the discretisation. Applying the thesis' conform finite element method for the Poisson equation, this error is in $\mathcal{O}(h^2)$ for a sufficiently smooth analytical solution. With the V-cycle, an algorithm needs a fixed number of iterations—independent of the number of unknowns—to reduce the solver error by a certain factor. It does not make sense to solve the equation system more accurately than up to the order of the discretisation error. Ergo, a grid refinement entails a fixed number of additional smoothing steps to get an error in the order of the discretisation error again.

A classical full multigrid algorithms starts on a rather coarse grid and computes a solution to the PDE with the well-known multigrid cycle—typically a V-cycle. Then, it projects this solution to the next finer level as initial guess for the new u_h and continues recursively. The resulting sequence of V-cycles on finer and finer grids is an F-cycle or FMG-cycle (Figure 4.12). An F-cycle solves a well-behaved problem in $\mathcal{O}(h^d)$, i.e. it depends linearly on the number of unknowns [73]. It is an optimal solver.

To make this hold, the projection of the current solution to a guess on the finer solution has to exhibit higher order. Higher order interpolation within the k-spacetree and the element-wise traversal world needs additional attention: The traversal automaton knows the solution's behaviour on a cell, as it has access to the 2^d vertices. For an interpolation of higher order, additional grid vertices had to be evaluated. This is usually not possible. Yet, due to an extension of the values stored within a vertex, a k-spacetree traversal is able to interpolate values with higher order. If the algorithm stores—besides the value u_h—the central differences within a vertex, the algorithm can reconstruct the $4^d - 2^d$ additional values adjacent to neighbouring cells. This allows to inscribe higher-order polynomials into the nodal fine grid approximation. In the implementation coming along with this thesis, the higher-order

interpolation is not yet integrated.

4.4 Traversal Events

The multigrid realisation plugs into the traversal events, and, consequently, the data available throughout the traversal has to be convenient for the multigrid solver. This restriction is accompanied and compensated by the possibility to merge different solver phases. In the following paragraphs, I analyse the data dependencies of the different solver steps (Figure 4.13). A mapping from traversal events to solver steps then enables the realisation of the FAS multigrid scheme. Let *active level* at this be the current smoother level. For two-level operations (restriction, coarsening and prolongation), the active level denotes the finer level. Furthermore, each variable in the linear equation system, i.e. each vertex $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ in the grid, holds five different properties/attributes: The weight of the shape function (the height of the hat), a residual variable to accumulate the residual, the right-hand side, the hierarchical surplus, and the hierarchical residual.

4.4.1 ω-Jacobi Smoother

The Jacobi solver (4.13) splits up into three implementation steps. All of them work on the nodal representation scheme.

First, the value of hanging vertices is interpolated. Peano's traversal preserves the child-father relationship. Within a refined element, the algorithm thus analyses the finer vertices throughout the steps down. If a vertex $v \in \mathbb{H}_{\mathcal{T}}$, and if its level is smaller than the active level, the algorithm interpolates the current solution. Hanging vertices beyond the active level remain unchanged.

Second, the algorithm evaluates the stencil. As the stencil covers 2^d geometric elements, this second step has to be split up additively. If a vertex of the active level is read for the first time, the multigrid solver sets its residual to zero. The residual is an additional attribute of the vertex. In each geometric element belonging to the active level, the solver evaluates the 2^d different stencils affecting the element's vertices and adds the result to their residuals. Before a vertex is written to the output stream, the residual variable holds $(b - A_\ell u_{\mathrm{h},\ell})_v$ as soon as the algorithm adds the right-hand side.

Third, the vertex is updated. According to the arguing above, a vertex's residual is available as soon as the vertex is to be written to the output stream: Here, the algorithm takes the residual and updates the current solution according to (4.13). Hanging vertices and vertices belonging to the boundary are not modified. Afterwards, the algorithm passes the vertex to the output stream.

Throughout the steps up, the realisation coarses the new solution of the active level. There is no numerical need for this coarsening, but, as a result, a coarse

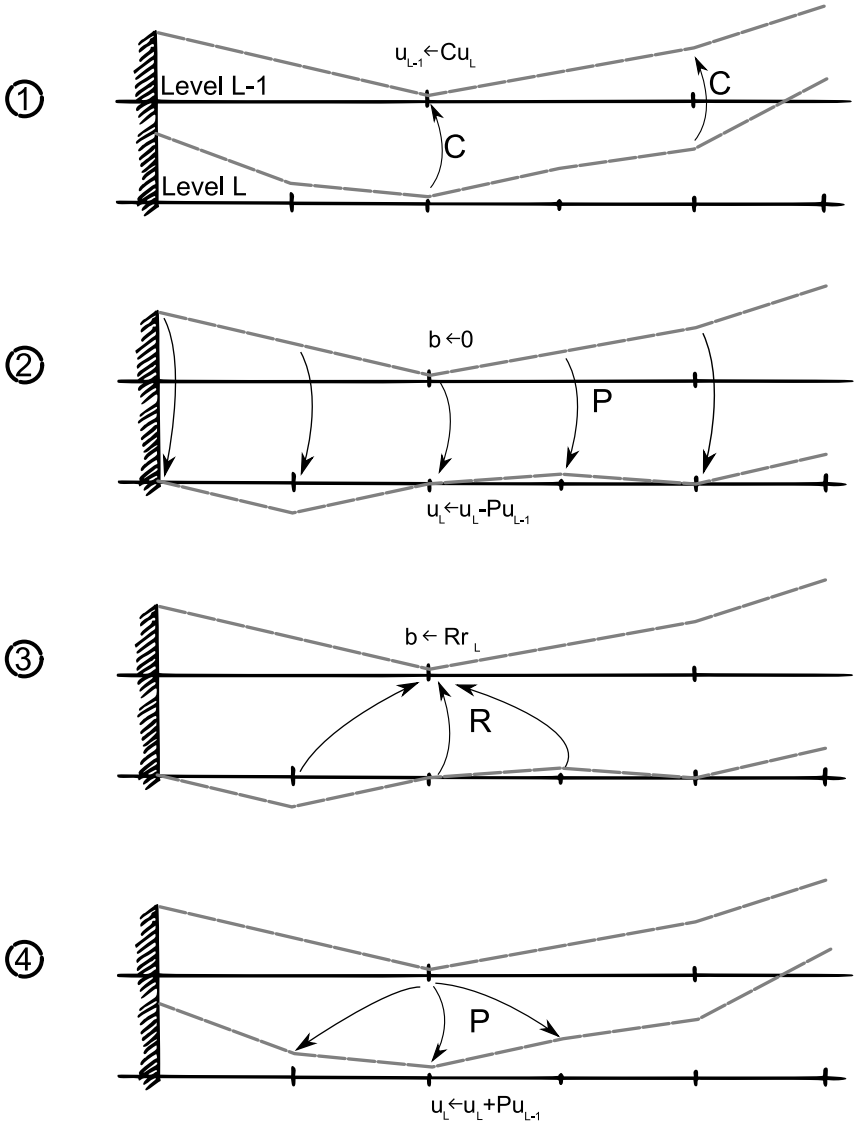


Figure 4.13: Behaviour of the hierarchical transform multigrid algorithm: Throughout the smoothing, the solution is always coarsened to all smaller levels (1). Throughout the ascent, the algorithm computes the hierarchical transform on the finer grid (2). The application of the stencil to the resulting hierarchical surplus yields the hierarchical residual on the fine level, and this residual is, in the same step, restricted to the coarse grid's right-hand side (3). The algorithm continues recursively. Throughout the descend, the algorithm adds the coarse grid's representation to the linear surplus stored on the fine grid (4).

grid representation of the solution is available on all levels besides the levels smaller than the active level. This is for example of value for on-the-fly visualisations of the solver's progress on a coarser level with reduced details. Furthermore, a coarsed representation is already available, if the multigrid decides to ascend in the subsequent iteration.

Most multigrid realisations rely on a Gauß-Seidel-type smoothers instead of the Jacobi scheme, since they exhibit better convergence and smoothing rates. Due to the information transport speed restriction, the solver implementation here can not implement a Gauß-Seidel.

Example 4.2. *Let vertex a and vertex b in $d = 2$ be neighbours on a regular Cartesian grid, and let the traversal implement a Gauß-Seidel smoother. A face connects a and b, i.e. there are two geometric elements e_1 and e_2 holding both a and b. Both contribute to the residual calculation for both vertices. A traversal handles element e_1 first, and the residual of a and b is incremented. It can not update one of the two vertices, as the contributions of element e_2 are still missing. The traversal continues. In geometric element e_2 it evaluates a part of the stencil affecting a. Now, let the residual of a be complete, i.e. the traversal updates the value of vertex a. Then, the traversal computes (new) a's contribution to b's residual. Though, the traversal computes only a's contribution due to e_2. The contribution due to e_1 can not be updated, as the traversal processes each element only once. Even worse, it already added a contribution to b with an invalid, old value of a, i.e. the traversal will end up with an overall invalid residual on b.*

The example shows that Jacobi is the natural choice for a smoother within the Peano world, as Gauß-Seidel-type solvers rely on a faster information transport speed than the element-wise traversal permits. Nevertheless, more sophisticated solver realisations exist. The Poisson equation on a staggered grid assigning the hats to the geometric elements instead of the vertices fits naturally to a Gauß-Seidel [51, 76]. In [23], some recursion unrolling techniques enable the user to implement a set of more sophisticated solvers such as hybrid Jacobi-Gauß-Seidel solvers, block Gauß-Seidel schemes on subdomains, red-black variants of Gauß-Seidel, and so forth.

4.4.2 Restriction and Hierarchical Transformation

The restriction of the right-hand side splits up into three steps: First, the hierarchical transform is computed. Afterwards, the grid holds the hierarchical surplus instead of the nodal value on the fine grid. Second, the algorithm determines the hierarchical residual. Third, this surplus is restricted to the coarser grid.

The right-hand side codetermines the residual, as it is added to the accumulated value (4.13). Since the algorithm's traversal preserves the inverse child-father relationship (Definition 2.2), all the 2^d adjacent elements $e \in adjacent(v)$ have been

traversed before, if $touchVertexLastTime$ is triggered for a vertex v. If they are refined, all their children have been processed before, i.e. whenever a vertex is contained within the support of v's shape function, $touchVertexLastTime$ has been called. An algorithm restricting a vertex's hierarchical residual throughout the $touchVertexLastTime$ event thus has a valid right-hand side for the coarse grid vertex v, whenever v's $touchVertexLastTime$ operation is invoked. As a result, my realisation merges the restriction process and the first smoothing iteration on the upcoming active level.

The algorithm computes the hierarchical transform throughout the step down events: If a vertex belongs to the active level, and if the vertex is not hanging, the algorithm takes the vertices from the level above and determines the linear interpoland's value at the vertex's position. Computing the linear interpoland is possible due to the local interpolation operators induced by the d-linear shape functions on the coarser grid. As the nodal value is redetermined throughout the inverse hierarchical transform, the variable's content holding the nodal value is replaced by the linear surplus. Hanging vertices get the value zero: they do not hold a shape function, and, thus, their linear surplus vanishes.

The algorithm accumulates the hierarchical residual throughout the traversal of the fine grid. As all vertices have been loaded whenever the automaton enters an element, all the vertices hold the hierarchical surplus due to the discussion in the paragraph above, and an application of the Laplacian stencil yields the local contributions for the hierarchical residual. The nodal residual on the active level is not needed throughout this traversal, since no solution update occurs, and I consequently store the hierarchical residual within the residual variable.

The algorithm restricts the hierarchical residual throughout the step up transitions: Whenever a non-hanging vertex belongs to the active level, it holds a hierarchical residual value stored in the residual attribute. Hence, the algorithm restricts this value to all the vertices of the father element that hold the refinement flag. If a coarse vertex does not hold the refinement flag, its right-hand side has been set during the vertex construction. It remains unaltered.

Besides the computation of the hierarchical residual on the fine grid, the algorithm also applies the Laplacian operator on elements belonging to the active level minus one. Here, the semantics of the residual variable remains the nodal residual, and it is actually used to update the solution.

4.4.3 Prolongation and Inverse Hierarchical Transformation

The prolongation of the coarse grid solution and the update of the fine grid approximation due to the inverse hierarchical transform consist of three steps: First, the coarse grid's nodal representation is interpolated to the fine grid. Second, the interpolated value is added to the hierarchical surplus stored within the fine grid's

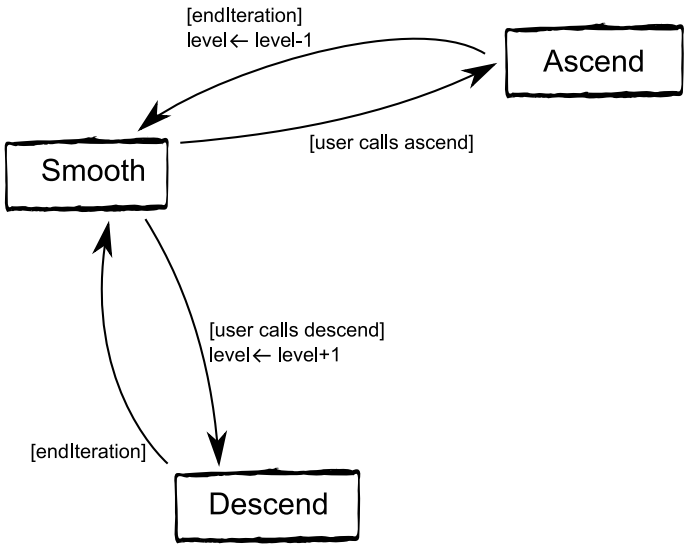


Figure 4.14: Different states of the multigrid solver.

u_v. Third, a fine grid smoothing step is applied to this nodal representation, i.e. one postsmoothing step is merged with the prolongation.

The two-level operations are embedded into the step down transitions. The algorithm analyses whether a fine grid vertex is adjacent exclusively to untouched faces. In this case, its inverse hierarchical transform has not been computed yet, and the vertex's solution attribute holds the hierarchical surplus. The solver computes the linear interpoland and adds it to this hierarchical surplus. As the algorithm applies a Jacobi smoothing step within the same traversal, hanging vertices are interpolated d-linearly throughout the steps down.

Besides the activities above, the standard Jacobi smoother actions from page 125 are invoked throughout the traversal. The traversal events hence are mapped to both the inverse hierarchical transform and the smoothing operations.

4.4.4 States of the Hierarchical Transformation Multigrid Method

The user steers the multigrid cycle with a state machine (Figure 4.14). Before each iteration, he tells the solver holding the active level whether to ascend or descend. If neither ascend nor descend are invoked, the solver realises a Jacobi smoothing step on the active level. Throughout the grid traversal, the solver's state is invariant.

The mapping from events to multigrid operations depends on the solver's state (Table 4.1): If the user triggers the ascend operation, the multigrid solver's state switches to `Ascend`. At the end of the iteration, the solver's state then switches back to `Smooth` and the active level is decremented. If the user triggers the descend

Table 4.1: Interplay of the traversal events, the solver states and the multigrid operations. All events not listed explicitly reduce to no operation. Let ℓ_{active} denote the active level, `First` abbreviates `Descend`.

Solver State	Traversal Event	Description and Operations
`Ascend`		Switch to the next coarser level throughout the iteration. Simultaneously, a smoothing step on this coarser level $\ell_{\text{active}} - 1$ is performed.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil. Accumulates the hierarchical residual $\hat{r}_v$. As the residual itself is not needed throughout the level's ascend, $\hat{r}_v$ is stored in r_v.
		$level(e) = \ell_{\text{active}} - 1 \Rightarrow$ apply stencil. Accumulates the residual in r_v.
	$touchVertexFirstTime(v)$	$r_v \leftarrow 0$, and
		$level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{refined}}(v) \Rightarrow b_v \leftarrow 0$.
		$level(e) = \ell_{\text{active}} \Rightarrow$ compute hierarchical transform.
	$createTemporaryVertex(v)$	$level(v) = \ell_{\text{active}} \Rightarrow u_v \leftarrow 0$, and
		$level(v) < \ell_{\text{active}} \Rightarrow$ interpolate coarse grid value.
	$touchVertexLastTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ restrict $\hat{r}_v$, with $\hat{r}_v$ stored in variable r_v.
		$level(v) = \ell_{\text{active}} - 1 \Rightarrow$ add right-hand side to residual variable and apply Jacobi update step.
`Smooth`		Apply Jacobi smoother on the active level.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil.
	$touchVertexFirstTime(v)$	$r_v \leftarrow 0$.
	$createTemporaryVertex(v)$	$level(v) \leq \ell_{\text{active}} \Rightarrow$ interpolate coarse grid value.
	$touchVertexLastTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ apply Jacobi update step.
`Descend`		Equals `Smooth`, but computes the inverse hierarchical transform before it applies the smoother.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil.
	$touchVertexFirstTime(v)$	$r_v \leftarrow 0$.
		$level(v) = \ell_{\text{active}} \Rightarrow$ compute inverse hierarchical transform.
	$createTemporaryVertex(v)$	$level(v) \leq \ell_{\text{active}} \Rightarrow$interpolate coarse grid value.
	$touchVertexLastTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ add right-hand side to residual variable and apply Jacobi update step.

operation operation, the multigrid solver's state switches to `Descend` and increments the active level. At the end of the iteration, the solver resets its state to `Smooth`. If the solver is told that this is the last traversal of the current V-cycle, it triggers both a Jacobi update on the active level, the computation of the local refinement criterion on each vertex and the evaluation of the global residual. The latter two aspects are discussed after some additional remarks on the solver's realisation and behaviour on adaptive grids.

4.5 Extensions and Realisation

With the preceding section, one can straightforward implement a full approximation storage scheme within the Peano traversal. A naive implementation though suffers from unbalanced spacetrees—the tree traversal always processes the whole tree although only a small part of the tree might belong to the active level and, thus, is actually updated. It is an obvious idea to alter the numerics and to adopt the scheme to strongly adaptive discretisations. The first subsection of the upcoming pages is dedicated to such a task. Next, I discuss which of the variables of a vertex have to be stored on the in- and output stream: the residual variable for example accumulates a helper value that is initialised at the first usage throughout traversal and evaluated before the data are written to the output stream. There is no need to hold such helpers persistently. Finally, I spend a few pages on a suitable yet simple refinement criterion—it compares the actual solution to a solution with doubled mesh width, i.e. studies the effect of one refinement step of h-adaptivity, and does not evaluate any complicated residual-based error estimates. Without such a criterion, the tuning to adaptive grids at the beginning of the section would be relevant solely to the domain's boundary, as regions within the computational domain were never refined.

4.5.1 Simultaneous Coarse Grid Smoothing on Adaptive Grids

Adaptive grids with areas of different resolution suffer from the set of rules in Table 4.1, as this set employs one global active level. If an area of the domain is tessellated by a coarser mesh than the mesh identified by the active level, the Peano algorithm traverses this part of the grid without performing any solution update. An improvement of the multigrid solver makes the solver simultaneously update the active level and all elements of the fine grid belonging to a level smaller than the active level. The smoother then equals a Jacobi on an adaptive grid.

The unrefined elements of a level $\ell < \ell_{\text{active}}$ identify a Cartesian grid where the solver can apply Jacobi updates. Updating both the active level and these coarser levels results in a solver working on different independent and uncoupled subdo-

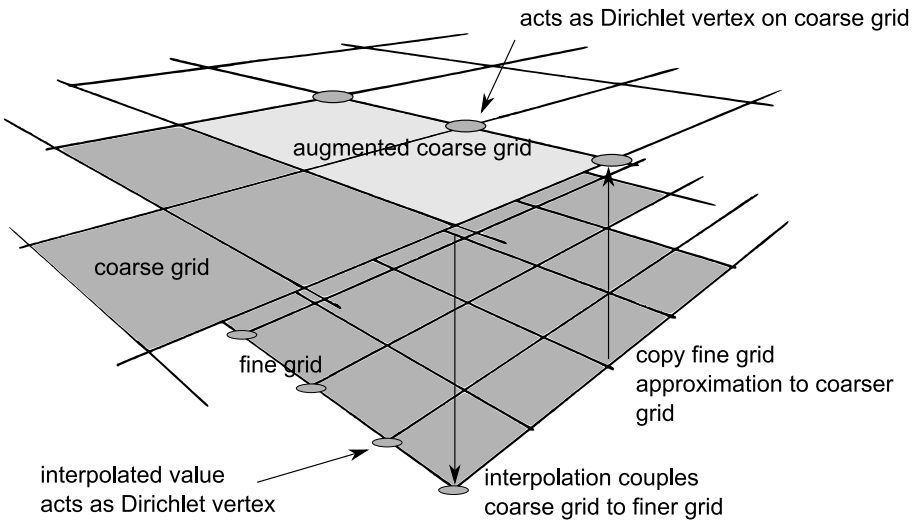


Figure 4.15: The simultaneous coarse grid smoothing extends the fine grid by one element wherever possible and applies a Jacobi on the resulting adaptive Cartesian grid. Interpolation and restriction couple the individual grid domains.

mains, i.e. it tackles the problem on ℓ_{active} and an additional number of small Dirichlet boundary problems on coarser grids.

To couple the individual problems, I first of all extend the Cartesian grid of each coarser level $\ell < \ell_{\text{active}}$ by one element, and, thus, introduce a shadow layer of width one around the domain. Boundary vertices of these individual grids are now either hanging vertices, refined vertices, or boundary vertices, and their values are prescribed: The hanging vertices' values result from the interpolation rules. If the solver improves coarser representations, the interpolated vertices change in the next iteration and the smoothing process on the finer grid continues with an improved interpolated value—interpolation transports information from coarse grids to finer grids. The refined vertices' values depend on the coarsening operator (4.7). Whenever the solver improves fine representations, the coarser vertices change in the next iteration and the smoothing process on coarser grids continues with an improved coarsened value—coarsening transports information from fine grids to coarser grids.

The implementation of the improvement is straightforward (Table 4.2) if a new predicate is introduced:

$$\forall e \in \mathbb{E}_{\mathcal{T}} : \quad \mathcal{P}_{\text{unrefined } v}(e) \Leftrightarrow \exists v \in vertex(e) : \neg \mathcal{P}_{\text{refined}}(v)$$

holds for all fine grid elements. Furthermore, it holds for all elements adjacent to fine grid elements. On levels smaller than the active level, it thus identifies all the geometric elements whose stencils have to be evaluated, and the resulting smoother scheme equals a domain decomposition approach where different Cartesian grids

Table 4.2: Modified rule set for the multigrid solver. For a given active level, it smoothes—besides the active level—also parts of the computational domain covered with a grid coarser than the active level. The rules replace entries from Table 4.1. Rules not enlisted again remain unaltered.

Solver State	Traversal Event	Description and Operations
`Ascend`	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil. $level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{unrefined } v}(e) \Rightarrow$ apply stencil. $level(e) = \ell_{\text{active}} - 1 \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ restrict $\hat{r}_v$, with $\hat{r}_v$ stored in variable r_v. $level(v) = \ell_{\text{active}} - 1 \Rightarrow$ apply Jacobi update step. $level(v) < \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step.
`Smooth`	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil. $level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{unrefined } v}(e) \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ apply Jacobi update step. $level(v) < \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step.
`Descend`	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil. $level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{unrefined } v}(e) \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ apply Jacobi update step. $level(v) < \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step.

overlap by at least one coarse element. The corresponding function basis holds one shape function per vertex. Yet, the Dirac property of the hats—$\varphi_v(v) = 1,\ \varphi_v(\hat{v}) = 0\ \forall \hat{v} \neq v$—does not hold anymore (Figure 4.15).

Such a modification of the event mapping yields a non-uniform smoother: The coarser an unrefined vertex is, the bigger the number of Jacobi updates for this vertex throughout every V-cycle. In the numerical experiments, the modified adaptive smoothing outperforms the standard multigrid, and there is hence no reason to skip the additional coarse grid updates—even if they might not be necessary from a smoother point of view, i.e. even if the error at these vertices is already sufficiently smooth—as long as the coarse subtrees are traversed anyway. This reasoning might not hold anymore, if the additional floating point operations slowed down the coarse grid traversal. I never observed such a behaviour.

4.5.2 Persistence and Semantics of Vertex Attributes

According to the introductionary remarks of Section 4.4, each vertex $v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$ in the grid holds five different values or variables, respectively. Yet, the hierarchical

surplus and the actual solution are never needed at the same time, and the algorithm can reconstruct one value from the other. Thus, both values are held by one variable having either the one semantics or the other. The same arguing holds for the residual and the hierarchical residual. This reduces the memory consumption by two floating point variables per vertex.

The (hierarchical) residual is accumulated throughout the iterations. The update uses this variable to improve the solution or to compute a new right-hand side whenever $touchVertexLastTime$ is called. In-between two iterations, the residual is not needed. The implementation hence does not store the residuals on the output and input streams. This reduces the memory consumption by one floating point variable per vertex.

4.5.3 Linear Surplus as Error Estimator

Peano's grid management is hidden from the PDE solver: the grid management triggers events, and the solver plugs into these events. Nevertheless, the communication is not unidirectional, as any event implementation is allowed to initiate a refinement or a coarsening on the vertices. The grid then refines or coarses, respectively, throughout the subsequent traversal.

In [21], the effect and implementation of different refinement criterion is evaluated. For the experiments here, I provide a refinement criterion based upon the *linear surplus*. Hereby, the algorithm compares the nodal solution u_v on the fine grid in each vertex to the mean value $\tilde{u}_v$ of the $3^d - 1$ surrounding vertices. If

$$|\tilde{u}_v - u_v| \geq \epsilon \qquad \text{or} \tag{4.20}$$

$$h^d \cdot |\tilde{u}_v - u_v| \geq \epsilon, \tag{4.21}$$

i.e. the nodal value exceeds a given threshold, the solvers refines v. (4.20) reduces the error in the $\|.\|_{max}$ norm, (4.21) reduces the error in the L_2 norm.

The mean value $\tilde{u}_v$ is modeled as additional variable for each vertex. Its value is set to zero within the $touchVertexFirstTime$. After that, the stencil evaluation accumulates it besides the residual. For $d = 2$, e.g.,

$$\frac{1}{8} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4.22}$$

computes the mean value $\tilde{u}_v$. It is available throughout $touchVertexLastTime$ where the algorithm evaluates the refinement formula and triggers the refinement for the next iteration. As for the residual, there is no need to store the mean value persistently.

The two simple error estimators above deliver a proof of concept that Peano's grid management, the full approximation scheme multigrid solver, and the dynamic

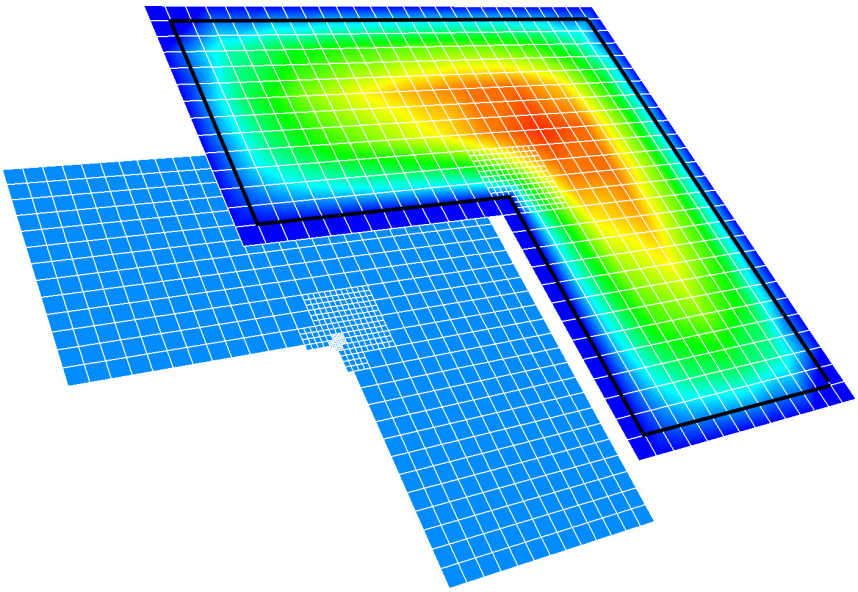


Figure 4.16: Adaptive grid for the L-shape with (below) and without (above) the boundary refinement criteria (4.23) and (4.24). (4.23) and (4.24) yield a finer grid around the semi-singularity, and they also highlight the boundary approximation improvement.

adaptivity fit together. Nevertheless, this thesis lacks an exhaustive discussion of different refinement criterion.

One remark is essential for the experiments: Criterion (4.20) and (4.21) observe only inner vertices. Typically, pollution and accuracy problems though stem from (semi-)singularities on the domain's boundary, i.e to reduce the mesh inside the domain does not resolve the root of the problem: Often, it only introduces additional hanging points on the boundary. Hanging points do not hold information, and, thus, not improve the boundary sampling (Figure 4.16). Consequently, I refine all vertices

$$\begin{aligned} v_{\text{boundary}} \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} : \quad & \neg \mathcal{P}_{\text{refined}}(v_{\text{boundary}}) \wedge \\ & v_{\text{boundary}} \in father(v_{\text{new}}) \wedge \\ & \mathcal{P}_{\text{boundary}}(v_{\text{boundary}}) \end{aligned} \tag{4.23}$$

whenever

$$\begin{aligned} & \mathcal{P}_{\text{inside}}(v_{\text{new}}) \wedge \\ \exists v' \in father(v_{\text{new}}) : \quad & \mathcal{P}_{\text{boundary}}(v') \wedge \\ \nexists v'' \in father(v_{\text{new}}) : \quad & \mathcal{P}_{\text{boundary}}(v'') \wedge \mathcal{P}_{\text{refined}}(v'') \end{aligned} \tag{4.24}$$

holds after the creation of a new vertex v_{new}.

A comparison of different refinement criterion for k-spacetrees is started in [21]. More elaborate schemes such as dual problem formulations there yield more appropriate grids for some problems, whereas the simple linear surplus delivers grids of sufficient quality for most examples while it is cheap to compute. For alternative Laplacian stencils, alternative mean value computations might result in better grids, i.e. in grids delivering the same quality of solution with a smaller number of grid points. The example above evaluates a mean value computation corresponding to a full stencil, as it evaluates $3^d - 1$ surrounding vertices. Other mean value computations such as five point rules evaluating $2d - 1$ vertices yield other adaptivity patterns. Besides the full mean value computation and the five point rule, the experiments also employ a skewed mean value stencil that exclusively evaluates vertices not connected to the mean value location by a hyperface. Also, the hierarchical surplus is an indicator how much precision one gains due to the refinement. Nevertheless, the hierarchical surplus proved to be an insufficient refinement criterion throughout the numerical experiments, as it introduces k-dependent refinement patterns. The counterpart of refinement is coarsening. A simple coarsening criterion also compares the mean value to the solution's value. It triggers a coarsening as soon as a coarsening threshold is underrun. Such a feature is of great value for time-dependent problems, e.g., where the grid of the preceding time step acts as startup grid for the subsequent time step.

4.6 Experiments

The following experiments illustrate the solver's numerical behaviour. The experiments concentrate on the multigrid properties, i.e. the figures enlighten the smoothing behaviour, the convergence rate, and so forth. They do not present runtime and memory requirements.

To measure the effect of a multigrid cycle, the experiments either compare two subsequent approximations' fine grid representations, or they measure the global residual r on the fine grid. Since the solver is defined on an adaptive grid, the norms take the mesh structure into account wherever necessary.

$$\begin{aligned} V &:= \{v \in \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} : \neg\mathcal{P}_{\text{refined}}(v), \\ \|u_{\text{h}}\|_{max} &= \max_{v \in V} |u_v|, \\ \|du\|_{max} &= \max_{v \in V} |u_v^{\text{old}} - u_v^{\text{new}}|, \end{aligned} \tag{4.25}$$

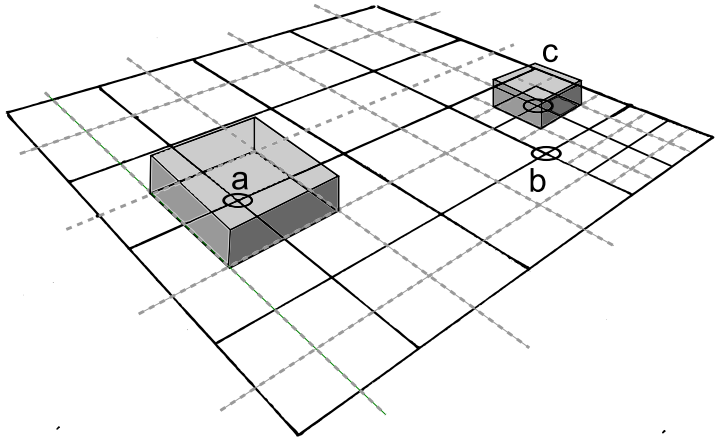


Figure 4.17: The $\|.\|_h$ norm interprets the nodal values as weights of a piece-wise constant nodal function basis where each basis elements surrounds one vertex (a). While it converges to the L_2 norm, it yields too big values for adaptive grids: Here, coarse "hats" (b) and finer "hats" (c) overlap.

$$\|u_{\mathrm{h}}\|_{\mathrm{h}} = \sqrt{\sum_{v \in V} h^d \left(u_v\right)^2},$$

$$\|du\|_{\mathrm{h}} = \sqrt{\sum_{v \in V} h^d \left(u_v^{\mathrm{old}} - u_v^{\mathrm{new}}\right)^2}, \tag{4.26}$$

$$\|r\|_{max} = \max_{v \in V} |r_v|, \qquad \text{and} \tag{4.27}$$

$$\|r\|_2 = \sqrt{\sum_{v \in V} r_v^2} \tag{4.28}$$

compute the discrete norms with h being the generic variable for the grid width of the surrounding geometric elements. The mesh width in (4.26) interprets the nodal value as weight belonging to a piece-wise constant ansatz space with each of its shape functions covering one fine grid element suitable dilated such that it surrounds the vertex. Thus, this metric corresponds to the L_2 norm of the measurand. If the grid is adaptive, the $\|.\|_h$ norm yields a value bigger than the L_2 norm, as the surroundings constant shape functions corresponding to the measurement overlap (Figure 4.17). This error vanishes with decreasing mesh width. Norm (4.26) is an extension of the $\|.\|_h$ norm in [10] to adaptive Cartesian grids. Obviously, the maximum norm (4.25) is independent of the grid structure.

In contrast to the solution's representation, the residual in (4.27) and (4.28) is by definition scaled with h^d due to the integral over Ω in the system matrix. It neither has an interpretation in the solution space nor is it an error indicator [10]. Hence,

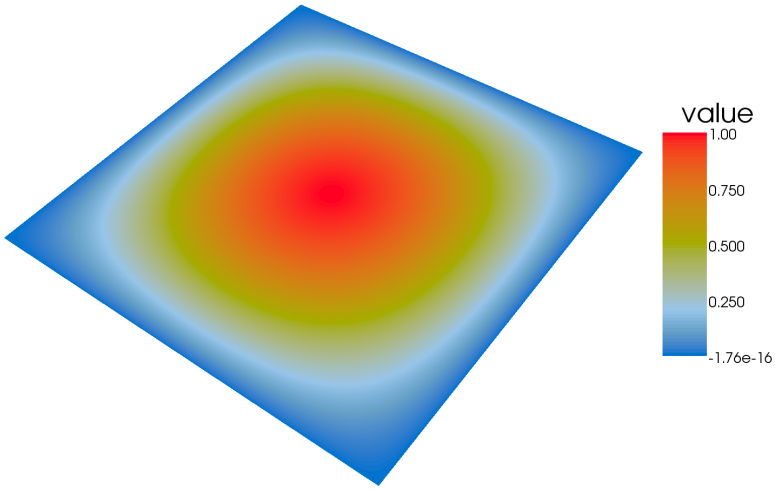


Figure 4.18: Solution of (4.30) for $d = 2$.

changes of the underlying grid change the semantics and order of the residual, too. Two residuals belonging to different grids thus are not comparable directly.

The *multigrid convergence factor*

$$\rho = \frac{\|e_{\mathrm{h}}^{\mathrm{new}}\|}{\|e_{\mathrm{h}}^{\mathrm{old}}\|} \tag{4.29}$$

for two subsequent iterations and suitable norms is measured with the residual

$$\rho \approx \rho_{\mathrm{h},\|.\|} = \frac{\|r^{\mathrm{new}}\|}{\|r^{\mathrm{old}}\|}$$

applying a suitable norm $\|.\|$. This approximation is accurate, if the grid is not changing anymore, and if the error is bigger than the discretisation error by a magnitude [73].

4.6.1 Convergence Rates

The first experiments solve the problem

$$\begin{aligned} -\Delta u &= d\pi^2 \prod_{i=1}^{d} \sin(\pi x_i), \\ u|_{\partial\Omega} &= 0, \qquad \text{with the analytical solution} \\ u &= \prod_{i=1}^{d} \sin(\pi x_i). \qquad (4.30) \\ \|u\|_\infty &= 1, \qquad \text{and} \\ \|u\|_{L_2} &= \left(\int_\Omega |u(x)|^2 dx \right)^{\frac{1}{2}} = \left(\prod_{i=1}^{d} \int_0^1 \sin^2(\pi t) dt \right)^{\frac{1}{2}} \qquad (4.31) \\ &= \left(\int_0^1 \frac{-1}{4} \left(e^{i\pi t} - e^{-i\pi t} \right)^2 dt \right)^{\frac{d}{2}} = \left(\frac{1}{2} \right)^{\frac{d}{2}}. \end{aligned}$$

The minimal and maximal mesh width are equal, and a dynamic refinement is switched off, i.e. the grid underlying the solution (Figure 4.18) is regular.

The smoothing behaviour of the Jacobi solver deteriorates with increasing mesh size, i.e. the finer the grid the smaller is the residual reduction per iteration (Figure 4.19). A $V(1,1)$-cycle in turn converges almost independent of both the grid size and the dimension (Figure 4.20). The maximum precision to be obtained depends on the discretisation error.

The corresponding $F(1,1)$-cycle is a cycle performing one $V(1,1)$-iteration per grid. It then adds the next level. This F-cycle yields results close to the V-cycle's output—the difference between the solutions is in the order of the discretisation error—but runs much faster, as the initial iterations traverse only coarse grids. Both multigrid schemes exhibit a convergence factor $\rho_{\mathrm{h},\|.\|} \in]4.69 \cdot 10^{-1}, 5.87 \cdot 10^{-1}[$.

The V-cycle's and the F-cycle's runtime profit from the horizontal tree cuts. With this feature, coarse grid update steps do not traverse the whole tree. They restrict to the upper part of the tree whose elements belong to the active level or a smaller level. As a result, the sum of the pre- and postsmoothing steps has to be even (Section 3.5.2). I return to absolute runtimes in the closing numerical results.

4.6.2 Multigrid Adjustment Screws

Multigrid algorithms rely on solvers that eliminate high-frequency errors. For the Jacobi smoother, the relaxation factor ω determines the smoother's effect on the different error frequencies. While the relaxation factors corresponding to bi-partitioning are well-studied, literature on arbitrary k-partitioning is rare. Nevertheless, the

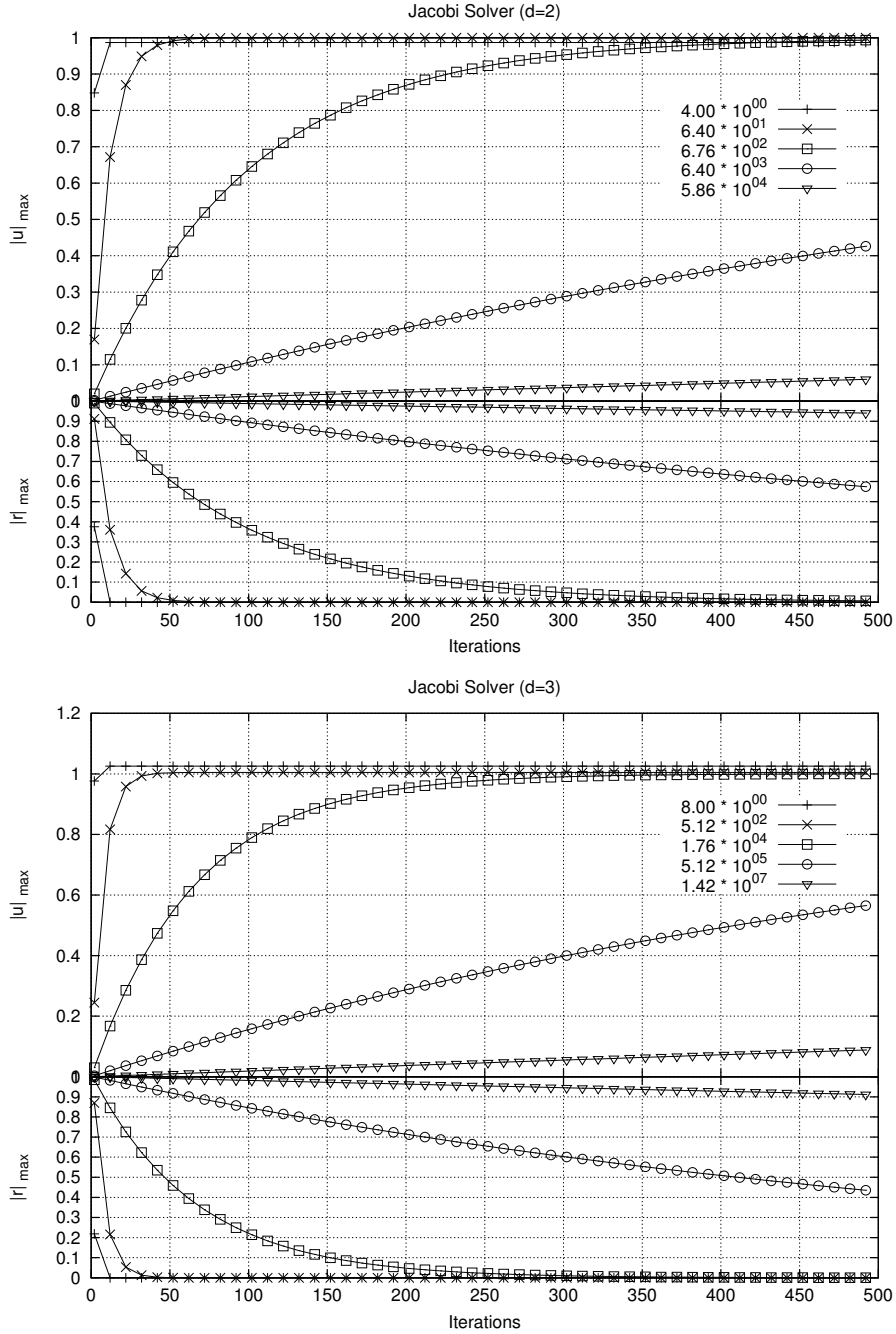


Figure 4.19: Behaviour of the Jacobi solver for (4.30): The residual is divided by the initial residual corresponding to the start solution guess $u = 0$.

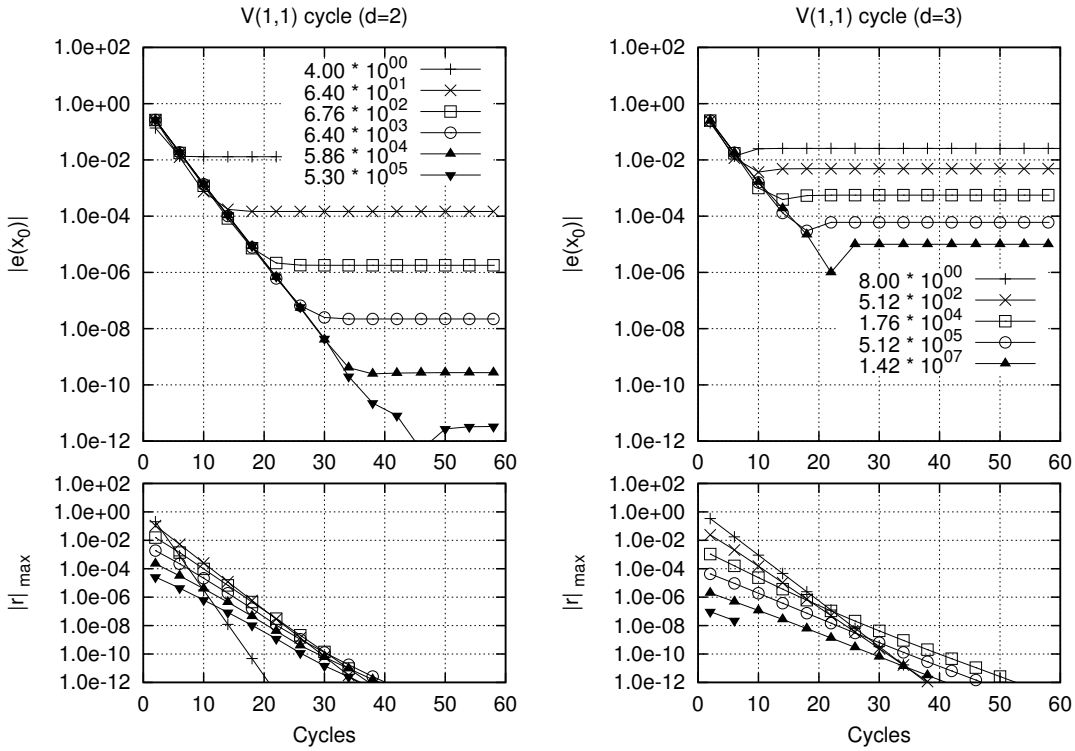


Figure 4.20: The multigrid solver tackles (4.30) with $V(1,1)$-cycles, and the error comprising both discretisations and solver contributions is measured at $x_0 = \left(\frac{1}{2}, \frac{1}{2}, \ldots\right)^T \in \mathbb{R}^d$. The legend denotes the number of unknowns.

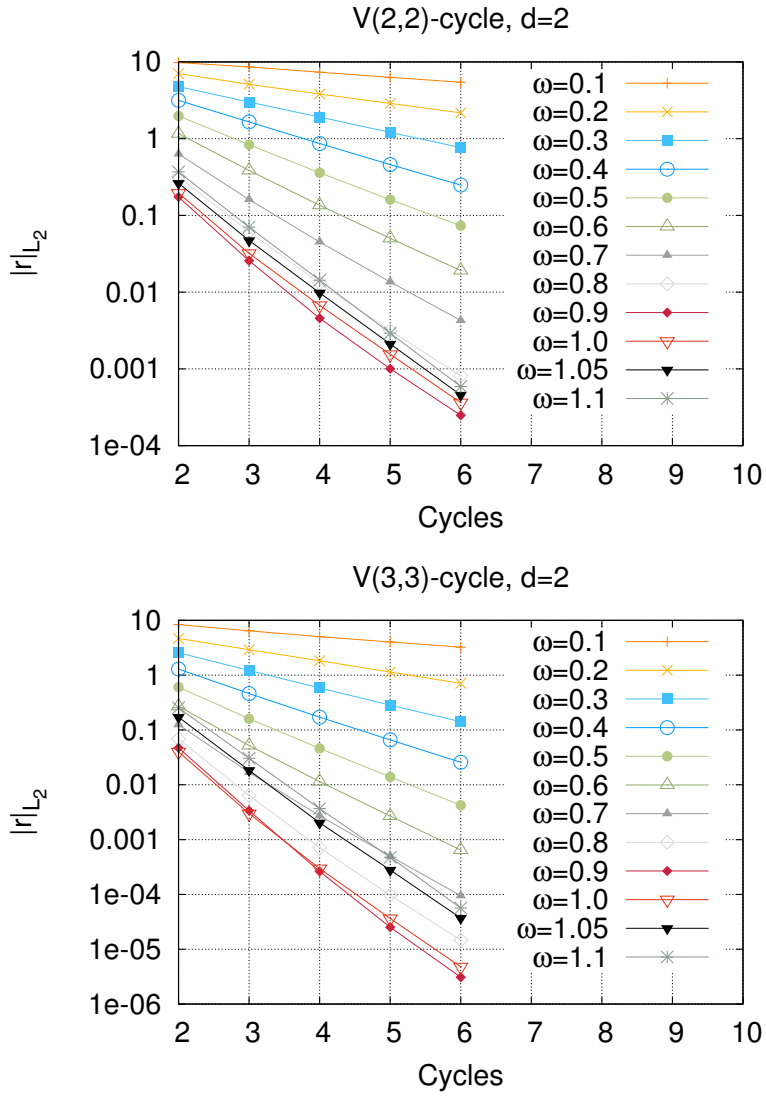


Figure 4.21: Influence of the relaxation factor ω on two different V-cycles.

Table 4.3: Residual $|r|_2$ for different V-cycles and $d = 2$. Regular grid for (4.30) with mesh size $3.3 \cdot 10^{-03}$, and $\omega = 0.8$. The F-cycle refines after each V-cycle until the grid meets a prescribed precision bound.

it	$V(2,2)$	$V(3,3)$	$F(2,2)$	$F(3,3)$	tree depth
1	$4.39 \cdot 10^{-04}$	$9.46 \cdot 10^{-05}$	$2.57 \cdot 10^{-02}$	$1.61 \cdot 10^{-03}$	2
2	$8.39 \cdot 10^{-05}$	$8.79 \cdot 10^{-06}$	$2.67 \cdot 10^{-01}$	$1.07 \cdot 10^{-01}$	3
3	$1.81 \cdot 10^{-05}$	$9.81 \cdot 10^{-07}$	$4.47 \cdot 10^{-02}$	$8.51 \cdot 10^{-03}$	4
4	$4.40 \cdot 10^{-06}$	$1.34 \cdot 10^{-07}$	$1.25 \cdot 10^{-02}$	$1.39 \cdot 10^{-03}$	5
5	$1.15 \cdot 10^{-06}$	$2.04 \cdot 10^{-08}$	$4.07 \cdot 10^{-03}$	$4.12 \cdot 10^{-04}$	6
6	$3.12 \cdot 10^{-07}$	$3.26 \cdot 10^{-09}$	$1.35 \cdot 10^{-03}$	$1.36 \cdot 10^{-04}$	7
7	$8.58 \cdot 10^{-08}$	$5.28 \cdot 10^{-10}$	$5.07 \cdot 10^{-07}$	$4.50 \cdot 10^{-08}$	7
8	$2.38 \cdot 10^{-08}$	$8.60 \cdot 10^{-11}$	$1.26 \cdot 10^{-07}$	$4.87 \cdot 10^{-09}$	7
9	$6.58 \cdot 10^{-09}$	$1.41 \cdot 10^{-11}$	$3.27 \cdot 10^{-08}$	$6.51 \cdot 10^{-10}$	7
10	$1.82 \cdot 10^{-09}$	$2.31 \cdot 10^{-12}$	$8.84 \cdot 10^{-09}$	$9.88 \cdot 10^{-11}$	7
11	$5.03 \cdot 10^{-10}$	$3.81 \cdot 10^{-13}$	$2.43 \cdot 10^{-09}$	$1.57 \cdot 10^{-11}$	7
12	$1.39 \cdot 10^{-10}$	$7.64 \cdot 10^{-14}$	$6.75 \cdot 10^{-10}$	$2.55 \cdot 10^{-12}$	7
13	$3.85 \cdot 10^{-11}$	$4.45 \cdot 10^{-14}$	$1.88 \cdot 10^{-11}$	$4.18 \cdot 10^{-13}$	7
$\rho_{\mathrm{h},\|\cdot\|_2}$	≈ 0.276	≈ 0.164	≈ 0.277	≈ 0.168	

choice of a relaxation is worth some effort, as the standard relaxation factor $\omega \approx \frac{2}{3}$ for bi-partitioning does not fit to $(k = 3)$-spacetrees (Figure 4.21). Instead, a relaxation factor $\omega \approx 0.9$ is appropriate. Following [41], an alternating sequence $(\omega_1 \approx \frac{1}{2}, \omega_2 \approx 1)$ of relaxation factors yields an even better rate. These results reoccur for $d = 3$, other combinations of pre- and postsmoothing as well as more complicated domains and right-hand sides. The experiments show that an elaborate Fourier and smoothing analysis here is yet to be done. This is beyond the scope of this work.

The fine-tuning of an F-cycle is a laborious work, as the number of pre- and postsmoothing steps of the underlying V-cycles can either be fixed a priori or made dependent on the residual's behaviour. Furthermore, their optimal choice also depends on the applied operators, the relaxation factor, and the smoother type. Since this chapter gives only a prove-of-concept, such a tuning is beyond the scope of this work. Table 4.3 nevertheless enlists some residual developments for different variations of pre- and postsmoothing steps, and compares it to the V-cycles' results.

The F-cycle exhibits a complicated behaviour (Table 4.3) splitting up into three phases: In the first phase (cycle one to six), the residual reduces monotonously as the grid is refined further and further. In the second phase (cycle seven), it diminishes by four magnitudes. In the third phase (from cycle seven on), the solver reduces the residual with an almost constant convergence rate.

For the big number of pre- and postsmoothing steps, each of the seven iterations ends up with a good approximation, i.e. the fine grid approximation is very accurate

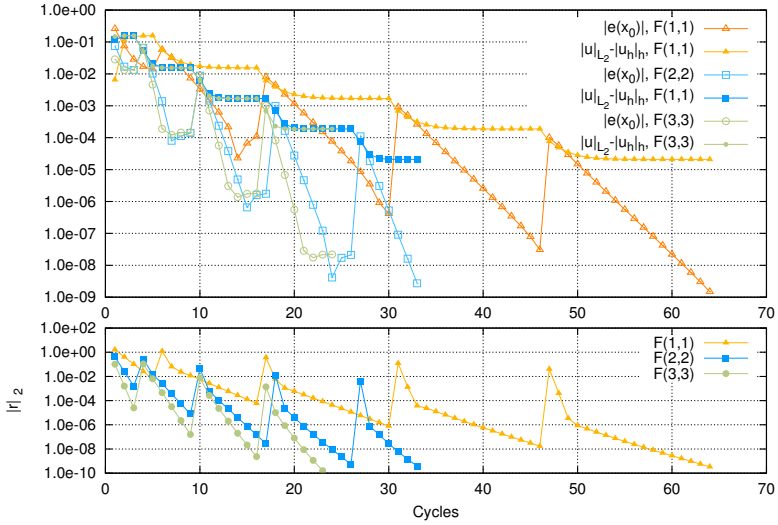


Figure 4.22: Experiment from Table 4.3. The F-cycle refines the grid each time the residual has been reduced by a factor of 100. Residual jumps identify these refinements.

whenever an additional grid level is added. The additional level resolves an additional high-frequency error. This high-frequency error is not tackled throughout the interpolation as the algorithm lacks the full multigrid's higher-order interpolation—it is the standard d-linear interpolation given by operator P. From cycle one to seven, the algorithm eliminates the coarse grid error and, in turn, adds an additional fine grid error. As soon as no additional level is added anymore, the residual is not added an additional high-frequency component, too. It "jumps" down. Afterwards, the solver behaves like a standard V-cycle. In [10, 73], e.g., the full multigrid algorithm typically exhibits a better convergence rate than the standard V-cycle, and there is no abrupt residual decay. Both facts illustrate the lack of a more sophisticated solver (Gauß-Seidel, e.g.) and a higher-order interpolation.

The influence of the lack of a higher-order interpolation is studied further in Figure 4.22: The F-cycle adds the subsequent grid as soon as the ratio of residual to initial residual underruns $1.0 \cdot 10^{-02}$. The experiment stops as soon as $\|du\|_{\mathrm{h}} < 10^{-12}$, and its figures illustrate the interplay of residual, grid levels, and actual solution. It reveals two insights: On the one hand, the peaks correspond to the cycles where an additional grid level is added. Their height should be reduced by a higher order interpolation. Nevertheless, the algorithm converges independently of the mesh width. On the other hand, it does not make sense for the F-cycle to wait for the V-cycle to converge. Instead, it can add another level immediately after one V-cycle.

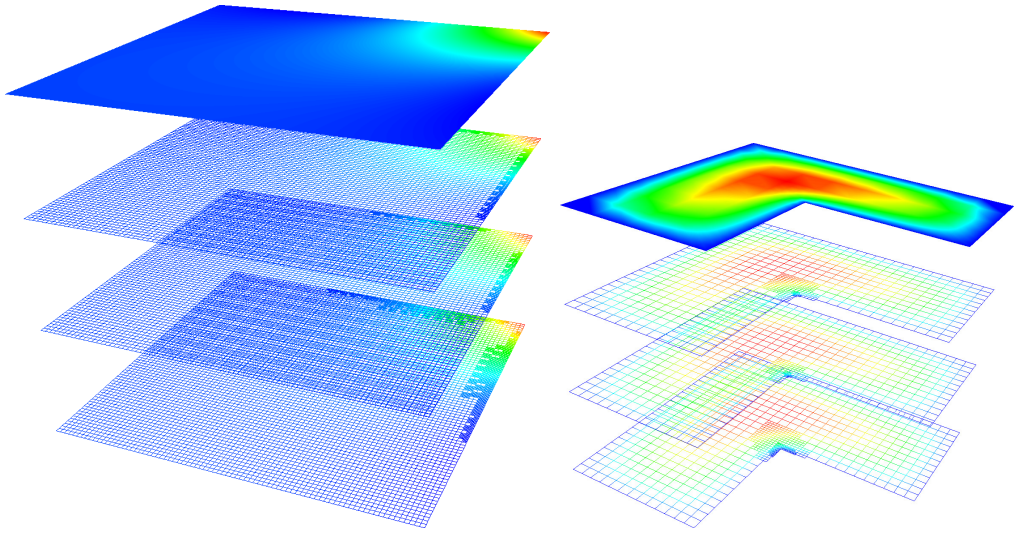


Figure 4.23: Experiment (4.32) with the solution, a grid resulting from the surplus stencil (4.22), a five-point stencil and the screwed five-point stencil (top down, left-hand side). Results for experiment (4.32) on the right-hand side.

4.6.3 Dynamic Adaptivity

In Section 4.5.3, the difference between the actual approximation and the mean value of the surrounding vertices acts as refinement criterion. Since this value vanishes for continuous solutions with a constant first derivative, the criterion measures the second derivative and refines where the second derivative is high. It assumes that such regions are of interest. The experiments studying this refinement strategy comprise, besides (4.30), the setups

$$\begin{aligned} -\Delta u &= 1, \\ u|_{\partial\Omega} &= 0 \qquad \text{with } \Omega =]0,1[^d \setminus \left]0, \frac{1}{2}\right[^d \qquad \text{and} \qquad (4.32) \\ u &= \frac{1}{\sinh(8\pi)} \cdot \cos\left(2\pi \cdot \left(\frac{x+1}{2} - \frac{y+1}{2}\right)\right) \cdot \\ &\qquad \cdot \sinh\left(2\pi \cdot \left(\frac{x+1}{2} + \frac{y+1}{2} + 2\right)\right) \\ &\qquad \text{with } \Omega =]0,1[^2. \qquad (4.33) \end{aligned}$$

The L-shape problem (4.32) exhibits a semi-singularity at $x = (\frac{1}{2}, \frac{1}{2}, \ldots)^T$, i.e. the first derivative there is not continuous, and the second derivative around this point is very large. The semi-singularity makes the solution not belonging to $\mathcal{O}(h^2)$ anymore:

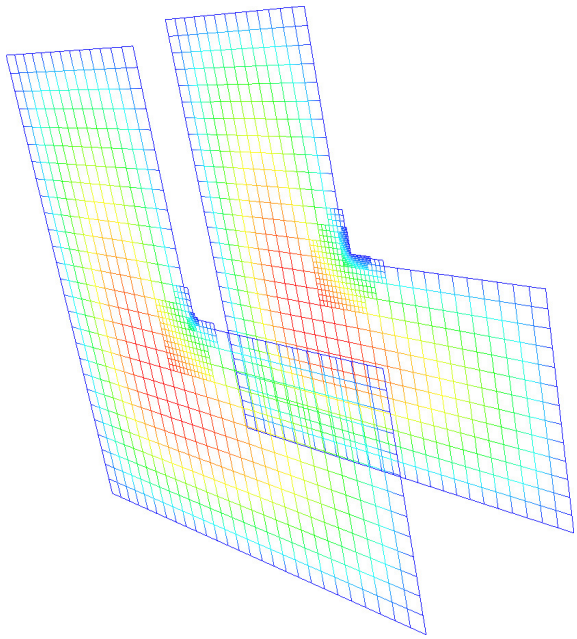


Figure 4.24: Grid of (4.32) with two linear surplus stencils: The full stencil's pattern (left) differs from the pattern of the skewed five-points stencil (right).

it introduces a pollution of the approximation. An adaptive grid refining around $x = (\frac{1}{2}, \frac{1}{2}, \ldots)^T$ eliminates this pollution.

Experiment (4.33) stemming from [64] is a smooth experiment, but its analytical solution exhibits a region of interest at the square's boundary around to $(1, 1)^T \in \mathbb{R}^2$. Here, the solution raises significantly, whereas in the remaining part of the domain the solution is almost constant.

Both experiments are illustrated in Figure 4.23. A zoom into the results in Figure 4.24 reveals the effect of different stencils to compute the mean value. The corresponding Tables 4.4, and 4.5 compare a $V(2, 2)$-cycle to a $F(2, 2)$-cycle for the three different experiments. Both experiments stop as soon as the solution difference underruns the current approximation by more than a factor of 10^{-10} in the $\|.\|_h$ norm, i.e.

$$\|du\|_h < 1.0 \cdot 10^{-10} \cdot \|u_{\mathrm{h}}\|_h. \tag{4.34}$$

The V-cycle works with a prescribed fixed mesh width inside the computational domain. No refinement criterion is switched on. The F-cycle starts on the coarsest grid containing at least one inner vertex. It then refines as long as the mean value error estimator (4.20) yields values greater than a prescribed threshold. It is chosen such that both the V-cycle and the F-cycle converge to a solution with the same number of accurate digits in the $\|.\|_h$ norm. The surplus is determined by the full mean value stencil. In accordance with the insights on page 144, the F-cycle does

Table 4.4: Comparison of a V-cycle to a F-cycle for (4.30). Both cycles terminate if (4.34) holds. $d = 2$ for the upper part, $d = 3$ for the lower section.

	$V(2,2)$			$F(2,2)$		
mesh width	vertices	depth	iterations	vertices	depth	iterations
$1.0 \cdot 10^{-01}$	$1.29 \cdot 10^3$	4	11	$1.29 \cdot 10^3$	4	13
$1.0 \cdot 10^{-02}$	$7.02 \cdot 10^4$	6	10	$1.52 \cdot 10^4$	6	15
$1.0 \cdot 10^{-03}$	$5.41 \cdot 10^6$	8	7	$1.66 \cdot 10^6$	8	10
$1.0 \cdot 10^{-02}$	$3.60 \cdot 10^4$	4	11	$3.60 \cdot 10^4$	4	13
$1.0 \cdot 10^{-03}$	$1.59 \cdot 10^7$	6	10	$2.34 \cdot 10^6$	6	15

Table 4.5: Comparison of a V-cycle to a F-cycle for (4.33). Both cycles terminate if (4.34) holds ($d = 2$).

	$V(2,2)$			$F(2,2)$		
mesh width	vertices	depth	iterations	vertices	depth	iterations
$1.0 \cdot 10^{-01}$	$1.28 \cdot 10^3$	4	15	$1.28 \cdot 10^3$	4	15
$1.0 \cdot 10^{-02}$	$7.02 \cdot 10^4$	6	17	$6.10 \cdot 10^3$	6	15
$1.0 \cdot 10^{-03}$	$5.41 \cdot 10^6$	8	16	$5.33 \cdot 10^5$	9	13

not search for a converged iteration for each individual grid refinement step, but evaluates and realises the refinement criterion after each V-cycle.

The results for (4.30) are enlisted in Table 4.4. Since the solution is smooth, the adaptivity criterion yields rather regular grids: the difference between the number of degrees of freedom for adaptive and regular grids is not significant, and both solvers work on a spacetree with the same maximum depth. The F-cycle needs some additional sweeps, as it first retreats to the coarser levels. These additional traversals might disappear with an interpolation of higher order implemented. Apart from the figures in the tables, the F-cycle nevertheless outperforms the V-cycle, as the complete F-cycle's initial iterations working on rather coarse grids are significantly faster than any V-cycle. If the number of cycle's required in total is (almost) constant, this setup phase dominates the overall runtime. This runtime observation is picked up again in the numerical conclusions. Finally, it becomes obvious that a termination criterion should adopt to the minimum mesh size for a real-world problem, i.e. the criterion should pay attention to the finest grid. For sufficiently smooth problems, a fine resolution (last measurement, $d = 2$) improves the coarse grid corrections, but in this example the solver terminates before fine grid solution "comes into play".

The results for (4.33) are enlisted in Table 4.5. The corresponding grid is the finer the farther away it is from the coordinate system's origin, as the solution's

Table 4.6: First ten cycles of a V-cycle and a F-cycle for problem (4.32)

	$V(2,2)$			$F(2,2)$		
iteration	vertices	depth	$\|.\|_h$	vertices	depth	$\|.\|_h$
1	$4.08 \cdot 10^6$	8	$6.47 \cdot 10^{-3}$	$2.63 \cdot 10^2$	3	$7.53 \cdot 10^{-3}$
2			$1.02 \cdot 10^{-2}$	$2.63 \cdot 10^2$	3	$1.09 \cdot 10^{-2}$
3			$1.25 \cdot 10^{-2}$	$1.19 \cdot 10^3$	4	$1.20 \cdot 10^{-2}$
4			$1.40 \cdot 10^{-2}$	$6.82 \cdot 10^3$	5	$1.34 \cdot 10^{-2}$
5			$1.52 \cdot 10^{-2}$	$5.32 \cdot 10^4$	6	$1.44 \cdot 10^{-2}$
6			$1.57 \cdot 10^{-2}$	$4.58 \cdot 10^5$	7	$1.52 \cdot 10^{-2}$
7			$1.62 \cdot 10^{-2}$	$5.68 \cdot 10^5$	8	$1.58 \cdot 10^{-2}$
8			$1.66 \cdot 10^{-2}$	$7.32 \cdot 10^5$	9	$1.63 \cdot 10^{-2}$
9			$1.69 \cdot 10^{-2}$	$9.08 \cdot 10^5$	10	$1.66 \cdot 10^{-2}$
10			$1.71 \cdot 10^{-2}$	$9.43 \cdot 10^5$	11	$1.69 \cdot 10^{-2}$

derivative increases with a growing distance. Thus, the refinement criterion yields a grid with a significantly lower number of vertices compared to a regular grid.

Finally Table 4.6 enlists the solution evolution for problem (4.32). Here, the adaptivity criterion yields a grid that is strongly refined around the boundary's concave vertex, and the F-cycle can compete with the V-cycles accuracy although it uses a grid that is significantly smaller.

4.6.4 Simultaneous Coarse Grid Smoothing

The multigrid suffers from adaptive grids, if the smoother acts exclusively on the active level. All the events on the fine grid cells belonging to a level that is smaller than the active level then degenerate to no-op. In Section 4.5.1 this rationale motivates Table 4.2. The underlying mechanism introduces a simultaneous coarse grid smoothing while finer grids are processed.

For (4.32), the F-cycle in the preceding section has to cope with a grid that is refined extremely around one single point. It is thus a good demonstrator for the effect of the simultaneous coarse grid smoothing (Table 4.7).

The coarse grid smoothing reduces the iteration numbers by more than a factor of two. This improvement becomes the bigger the higher the more accurate the refinement criterion. The traversal always runs through the whole input stream, and the simultaneous coarse grid smoothing does not alter its cardinality. Hence, the runtime depends linearly on the reduction of cycles. Adaptive grids benefit from the simultaneous coarse grid smoothing. The improvement vanishes for regular grids. A runtime analysis is to be found at the end of the thesis.

Table 4.7: Number of iterations for an F-cycle on problem (4.32) with coarse grid smoothing. Both experiments stop if $\|du\|_h \leq 1.0 \cdot 10^{-5}$. Either, the solver exclusively processes the active level (off), or it applies the rules from Table 4.2, i.e. all fine grid levels smaller or equal the active level are updated (on).

refinement	$d = 2$		$d = 3$	
criterion threshold	off	on	off	on
$1.0 \cdot 10^{-01}$	38	22	37	22
$3.3 \cdot 10^{-02}$	42	21	41	21
$1.0 \cdot 10^{-02}$	45	19	44	19
$3.3 \cdot 10^{-03}$	43	18		

4.7 Outlook

While every PDE beyond the Poisson equation brings along its own challenges and difficulties, an efficient solver for the Poisson equation and the accompanying insights and rationale are a good starting point for any solver. They show that sophisticated multiscale algorithms fit to the k-spacetree world. Some projects already exploit and extend this chapter's principles: The ingredients on hand for a collocated degree of freedom layout (the unknowns are assigned to the vertices) can be transferred to a staggered, i.e. cell-centered, layout where each geometric element corresponds to an unknown [51, 76]. Such a discretisation of the function space is fundamental for many computational fluid dynamics codes [59, 60]. And while $d \in \{2, 3\}$ is a natural choice for the Poisson equation, the k-spacetree idea allows arbitrary d, and it is an obvious idea to develop schemes for instationary equations with a full space-time discretisation.

The insights coming along with the experiments are twofold. On the one hand, the results show—besides the missing higher-order interpolation—the approach realising all the features one expects from an up-to-date solver: The convergence behaviour is independent of the mesh size, the adaptivity is able to resolve pollution effects resulting from singularities, the memory requirements are low, and grids permanently changing throughout the computation do not pose a restriction on the solver's behaviour. Both the low memory demands and the latter aspect are due to the fact that no stiffness matrix is set up.

On the other hand, the convergence rates are low compared to literature. The solver is not competitive. As the multigrid behaviour holds, the low convergence rates are due to a poor smoother. To end up with a competitive iteration behaviour, the multigrid has to use a more sophisticated solver than a Jacobi. Unfortunately, the choice of this particular smoother results from the speed information

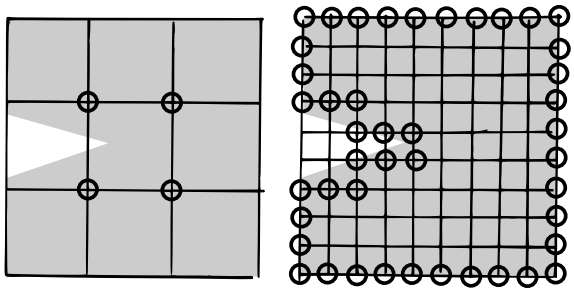


Figure 4.25: Domain with "complicated" shape. The marked vertices are boundary vertices, i.e. they do not hold a shape function and a degree of freedom. The multigrid solver degenerates to a Jacobi solver as there are no shape functions to implement a coarse grid correction.

can be transported by the element-wise k-spacetree traversal (Section 2.6). Nevertheless, there are better solvers available within the Peano framework: a Gauß-Seidel type solver can be realised within the framework, if the discretisation switches to a element-centered degree of freedom layout [51, 76] which is a popular choice for example for the pressure Poisson equation in computational fluid dynamics. For this, the choice of an odd k proves of value: Every coarse grid cell's center coincides with a fine grid cell center. If this would not hold—any even k implies this—coarser grids would end up with unknowns assigned to both vertices and elements. For an odd k, all grid levels exhibit a homogeneous staggered degree of freedom layout.

Besides the improved smoothers, two further aspects have to be mentioned in the context of k-spacetrees. First, singularities typically yield grids that are refined extremely around the singularity, i.e. each grid refinement step entails several refinement steps at the singularity. Within the multigrid solver context with simultaneous coarse grid smoothing, the effect reoccurs the other way round: The solver starts with an active level set to the maximum level. Every time the active level is decremented, the grid also could switch to a coarser level in the coarse regions of the solution, as the solution there is sufficiently smooth, too. A local active level equals such an approach. This discussion picks up the local tree cut paradigm (see Figure 3.21), and both concepts have to be combined in future work.

Second, geometric multigrid algorithms suffer from irregular domains. As the continuous domain shrinks to the individual grid levels, non-smooth domain boundaries make the coarse grid representations significantly smaller than the finer grids. Then, they hold a significantly smaller number of shape functions (Figure 4.25), and the coarse grid correction effect deteriorates. An improved boundary handling tracking the actual intersection of the continuous domain with the geometric elements resolves this problem [75]. Boundary vertices then hold a shape function, but this shape function is tailored to the actual domain boundary. As a result, the coarse

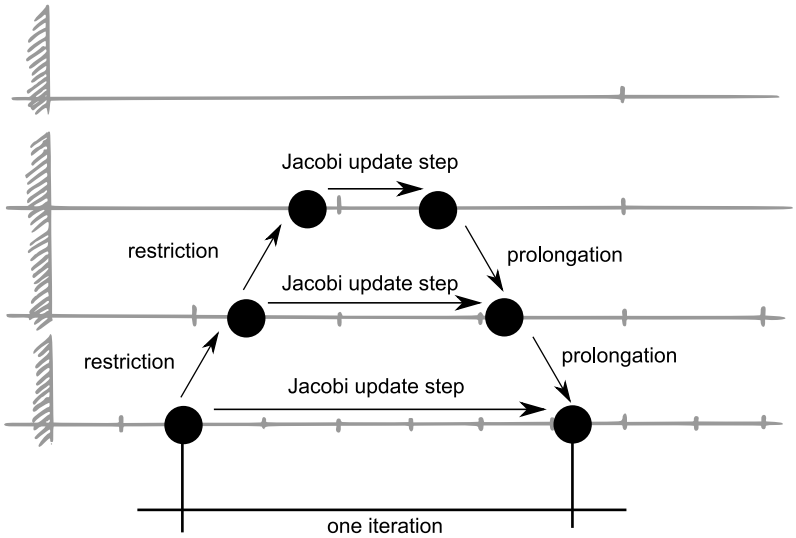


Figure 4.26: Additive multigrid scheme: The solver computes a fine grid residual, restricts this residual immediately to all levels and updates all variables on all levels. Before the next iteration, a prolongation (and summation) of all the level contributions yields the next iteration's solution.

grid correction effect improves. Furthermore, the boundary approximation accuracy improves to $\mathcal{O}(h^2)$, which is especially important for $d > 2$, as Section 2.7 points out.

4.7.1 Additive Multigrid

The forerunners of this thesis [35, 39, 63] also solve the Poisson equation, i.e. they realise exactly the same demonstration challenge. Their implementations provide an additive multigrid scheme restricting the fine grid residual simultaneously to all grid levels. The Jacobi update step then updates all grid simultaneously, too, and these update steps are combined to a fine grid solution throughout the successive depth-first traversal (Figure 4.26).

Additive multigrid algorithms suffer from the fact that the the finest level has to be processed for each iteration. They are thus slower than multiplicative schemes for most problems. Nevertheless, there are setups where the additive variant is more robust.

As the multiplicative scheme here is based upon a depth-first traversal, it is possible to replace the naive Jacobi smoother with the additive multigrid scheme. The result is a multiplicative multigrid scheme with a Jacobi smoother that is preconditioned with an additive multigrid. Such an approach sounds promising.

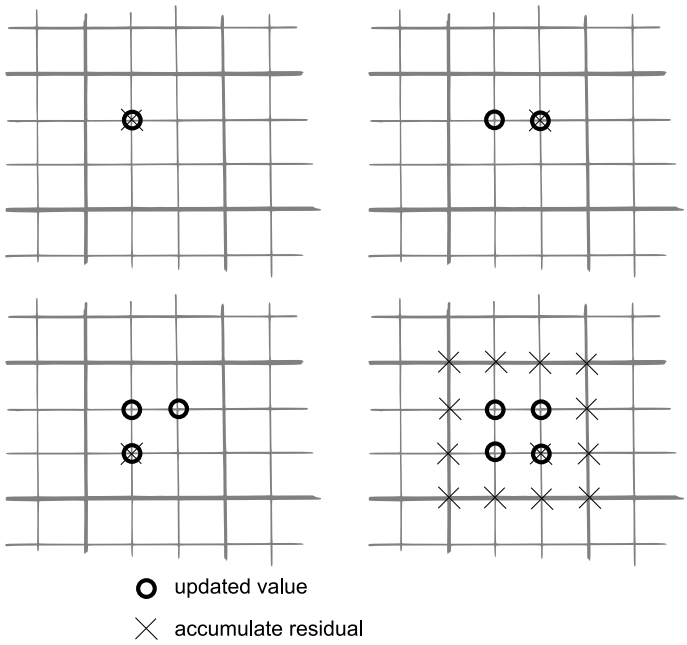


Figure 4.27: Block Gauß-Seidel: If a 3^d patch's values are available, the algorithm can update one vertex and immediately use the updated value to update the other vertices. Within the block, it is a Gauß-Seidel scheme. At the boundary, the residuals are finally accumulated. Here, it remains a Jacobi smoother.

4.7.2 Block-wise Smoothers

Space-filling curves are the fundament of an efficient realisation of the traversal in Chapter 3. Efficiency refers to both the memory requirements and the memory access characteristics. Besides these two factors, they also proof of value within the parallelisation context in the upcoming chapter.

Nevertheless, first experiments with the running code highlight a fundamental challenge coming along with the sophisticated traversal order: The traversal consists of lots of code and lots of integer arithmetics which finally even might slow down the floating point performance. If the code unrolls recursion steps, small blocks of the adaptive Cartesian grid can be processed as regular Cartesian blocks. Such an unrolling leads to the parallelisation approach in the upcoming section, and it allows for sophisticated algorithmic optimisations [23].

Besides optimisation and parallelisation, recursion unrolling enables the solver to handle small sections of the Cartesian grid as regular grid block. In these blocks, the information transport restriction for the smoother does not hold anymore, and the solver can employ a more sophisticated solver. If the code merges for example k^d blocks—Chapter 5 actually establishes this merging—the code can update the

vertices within these blocks with a Gauß-Seidel scheme, e.g. (Figure 4.27), whereas the blocks' boundary update scheme remains a Jacobi scheme. Such an algorithm equals a block Gauß-Seidel smoother. Extending this idea to bigger patches of unrolled recursion steps establishes the basis for a set of smoother improvements: (Multicore) Gauß-Seidel schemes and red-black schemes become realisable. Several iteration steps can be merged [48] or matrices belonging to the patches can be inverted explicitly and, thus, whole subblocks are solved exactly.

5 Parallelisation

Parallelisation is crucial for simulation codes. At least three observations confirm this statement: First, new computer architectures gain computing power due to an increased level of parallelism. While their clock rate and instruction complexity remain roughly the same from generation to generation, the number of computing units per computer raises, and any application profiting from future architectural improvements has to exploit multiple computing units. As a result, Moore's law still holds, but it does not apply directly to the (single node) performance [71]. Second, new simulation codes produce more and more data, i.e. the amount of data rapidly exceeds a single node's memory. Scientific insights however rely on an increasing level of detail. Third, new data is of value if and only if it is delivered within a reasonable time. Consequently, the convenience of a simulation workflow depends on the latency data is delivered by a simulation code. The data problem is tackled in Chapter 3. Nevertheless, reducing the memory spent on a PDE solver postpones the data threshold, but it does not resolve the underlying problem. The Peano framework consequently has to run in parallel.

Domain decomposition methods are the predominant approach to parallelise multiscale PDE solvers and make them keep step with the increasing architectural concurrency [45]. They split up the computational domain into several overlapping or non-overlapping parts (see [54], e.g., for some remarks and citations on multigrid solvers with different overlapping and non-overlapping domains), and they distribute these parts among the computing nodes. The problems on the individual nodes are small compared to the original problem, i.e. they can be solved faster with less memory. k-spacetrees induce a multiscale representation of a computational domain, and each level already represents a non-overlapping—I consider exclusively non-overlapping approaches here—decomposition of the domain into geometric elements. A mapping from the geometric elements to computing nodes yields a domain decomposition. It is balanced, if all computing nodes are assigned the same workload.

For multiscale domain representations where not only data but data with a hierarchical topology are to be split up, two mapping paradigms compete: Some algorithms analyse the fine grid and distribute the fine grid's elements among the computing nodes. These elements then determine to which node coarser elements belong to: Whoever holds fine grid elements is a candidate to handle their father elements, too. If there are multiple candidates, some approaches duplicate the fa-

ther responsibilities ([50, 55], e.g.)—such an overlap is sometimes referred to as *full domain partitioning*—while others employ a decision function selecting one dedicated node ([18, 19], e.g.). This is the way followed here. Technically, the partitions overlap slightly, but logically they are disjoint. Both realisation variants follow a bottom-up approach balancing the workload on the fine grid. Balancing the fine grid facilitates a fine granularity due to the big number of atomic grid constituents, while coarse partitionings depend on the fine grid partition. Operations on the coarse partitions might not scale. Alternative algorithms distribute a fixed coarse grid among the computing nodes ([26, 29], e.g.). Descendants of one coarse grid element are then processed by the same computing node. It is a top-down approach balancing the work on the analysed coarse grid. This partitioning is usually comparably cheap to determine. If finer grids result from a uniform refinement, their balancing is good, too. Otherwise, fine grid distributions might be unbalanced.

Whenever a data decomposition is realised, it either distributes the data exclusively among the computing nodes, i.e. each node has access to its and only its data chunk, or it is based on a shared, common data set. Latter realisations rely on shared memory architectures: it is important that each computing node can access any data any time. Implementations splitting their data repositories are more general and fit to both shared and distributed memory architectures. As they have to synchronise and preserve the distributed data consistency manually, they though come along with an additional overhead. This discussion restricts itself to distributed memory parallelisation realised by a message passing interface. It thus fits to both shared memory or distributed memory clusters. To exploit a shared memory architecture without the additional overhead introduced by a logical splitting of the data working set is beyond the scope of this work. Extensions of this thesis nevertheless adopt the Peano framework to shared memory architectures ([23], e.g.).

This chapter's ideas mirror the recursive top-down traversal, i.e. they start balancing the workload top-down throughout the grid setup on each single level. Later throughout the simulation run, the fine grid workload is measured bottom-up, and the partitioning/balancing is adopted and optimised. However, with a k-spacetree where individual subtrees are assigned to remote nodes, a depth-first approach is useless as it descends into subtrees sequentially. I thus combine the Peano traversal with a single-step breadth-first order into the level-wise depth-first order and, herefrom, introduce a recursive master-worker topology on the computing nodes, i.e. the computing nodes are organised as a tree themselves. While the partitioning is based upon ideas of [31, 40, 50], the combination of a hierarchical assignment with the parallel Peano traversal is—to my knowledge—a new approach to exploit a k-spacetree for a domain decomposition.

As soon as a parallel traversal for a given partition is available, the question arises what a good partition looks like. A good partitioning is at hand, if the workload is balanced, and if the surface-volume ratio of the individual partitions is small: the

amount of computations per node then is big compared to the amount of data to be exchanged, as the computational load corresponds to the volume of a subdomain, whereas the exchanged data volume corresponds to the surface of the subdomain. Partitions induced by space-filling curves are well-known in the parallel community for their good surface-volume ratio [17, 31, 42, 44]. Consequently, Peano's parallelisation benefits from the Peano traversal, although the decomposition and load balancing ideas are not correlated with the space-filling curve. To balance the workload, the Peano framework applies a bottom-up cost model to the k-spacetree and derives a top-down optimality metric from this. The model incorporates both the fine grid workload and the multi-level representation of the grid. Such an approach equals a tree attribute grammar [46] formalising a discrete optimisation problem on an acyclic connected graph. It quantifies the balancing.

As Peano's grid is allowed to change throughout the computation, the algorithm has to adopt the parallel partitions permanently. Again, different approaches are available for such a dynamic load-balancing: My decomposition approach restricts the balancing activities to forks and joins, i.e. a partition can either split up into several partitions, or several partitions can be merged. Such an approach fits to the master-worker paradigm mentioned before, as only new master-worker relationships are established or existing masters carry over grid parts from their workers. Workers released by their masters can be re-employed by other nodes. The load balancing permanently compares the nodes with the heaviest workload to the laziest nodes. Whenever a node could afford to do the work of one of its workers, it merges with this worker. Whenever a node slows down the computation, it hires additional workers. The approach is a greedy algorithm based upon the tree attributes, and it is decentralised, i.e. the balancing itself runs in parallel and does not become a sequential bottleneck. The parallel community offers a vast range of different parallelisation paradigms, algorithms, and taxonomies. While it is hard to keep step with all the ideas coming up—many problems need a tailored solution of their own—it is almost impossible for one thesis to give a comprehensive overview and classification of the own ideas. I thus concentrate exclusively on Peano's approach and leave it to the reader to classify and compare it to other solutions.

The chapter is organised as follows: At first, the parallel traversal for a given partitioning of the k-spacetree is established (Chapter 5.1). As the sequential depth-first Peano traversal does not fit to the partitioning approach, I extend it. Next, Section 5.2 picks up the fact that space-filling curves, on the one hand, imply quasi-optimal parallel partitions. On the other hand, they simplify the exchange of the data assigned to the vertices on the domain boundaries. Tree attributes quantifying a partitioning are introduced in Section 5.3. Two predicates triggering the forks and joins then derive herefrom. Realisation details—the mapping of the computational nodes to the hardware, the concept of a node pool, the buffering of messages, and so forth—in Section 5.4 lead over to a discussion of the interplay of the parallelisation

and the Poisson solver from Chapter 4. Some numerical experiments and a short outlook close the chapter.

5.1 Parallel Spacetree Traversal

In a domain decomposition algorithm, each node has to traverse its local domain, and partition boundary operators and the exchange of the subdomains' boundaries ensure that the parallel code preserves the sequential semantics of an algorithm. Typically, the traversal start and the traversal end act as synchronisation points. A decomposition then defines a partial order on the computations, as the traversals are ran one after another, but the computations throughout the iteration run in parallel.

All the algorithms in this thesis are based on an element-wise traversal (Section 2.4) of the k-spacetree. Though the semantics of many algorithms such as the multigrid approach in Chapter 4 does not rely on the actual order of the k-spacetree's nodes—they depend solely on the child-father relationship—it proves of great value to arrange the individual nodes within the depth-first traversal along the iterates of the Peano space-filling curve (Section 3.3). The resulting element-wise traversal defines a total order on the nodes of the k-spacetree.

Any total order prohibits a straightforward parallelisation, since each parallelisation exploits the freedom of a partial evaluation order. A total order lacks this degree of freedom, i.e. a depth-first traversal $\sqsubseteq_{\text{dfo}}$

$$\begin{aligned} a \sqsubseteq_{\text{child}} b &\Rightarrow b \sqsubseteq_{\text{dfo}} a, \\ c \sqsubseteq_{\text{pre}} d &\Rightarrow c \sqsubseteq_{\text{dfo}} d, \qquad \text{and} \\ e \sqsubseteq_{\text{child}} f \wedge f \sqsubseteq_{\text{pre}} g &\Rightarrow e \sqsubseteq_{\text{dfo}} g, \qquad a,b,c,d,e,f,g \in \mathbb{E}_{\mathcal{T}}, \end{aligned} \tag{5.1}$$

with $\sqsubseteq_{\text{pre}}$ giving an order on the children of each refined element is not well-suited for a parallel algorithm. $\sqsubseteq_{\text{dfo}}$ here specifies how geometric elements are read the first time. The store order, in turn, results from the recursive implementation of the traversal.

Example 5.1. *Let $k = 3, d = 2$ with $a_1, a_2, \ldots, a_9 \sqsubseteq_{\text{child}} a_0$ and $a_i \sqsubseteq_{\text{pre}} a_{i+1}$, $i \in \{1, \ldots, 8\}$. a_1 and a_2 are refined. A Peano traversal can not deploy the work on a_1 and a_2 to different nodes, as a_1 and all its descendants have to be processed before a_2 according to the depth-first total order.*

To make the spacetree traversal run in parallel, one has to weaken the processing order into a partial order. Three observations accompany this idea:

1. The overall child-father relationship has to be preserved. Otherwise, one has to rewrite the complete multigrid and stack access algorithms, as both concepts rely on this relationship.

2. The order in which the events are called on the children of one refined element does not affect the algorithm's behaviour because of the restriction on the information speed (see Section 2.6). Here, I can weaken the order and parallelise the individual events.

3. The order of the children of one refined element has to preserve $\sqsubseteq_{\text{pre}}$ such that the stack-based grid management still works.

It is not possible to trigger the events of observation two in parallel within a depth-first traversal, as a depth-first traversal never provides all the data of the k^d children of one refined element at one point. I hence elaborate a modified traversal facilitating such a parallelism.

Definition 5.1. *The* level-wise depth first order $\sqsubseteq_{\text{lw}}$ *is an order on a k-spacetree with*

$$a \sqsubseteq_{\text{child}} b \quad \Rightarrow \quad b \sqsubseteq_{\text{lw}} a,$$

$$c \sqsubseteq_{\text{pre}} d \quad \Rightarrow \quad c \sqsubseteq_{\text{lw}} d, \tag{5.2}$$

$$e \sqsubseteq_{\text{child}} f \wedge f \sqsubseteq_{\text{pre}} g \quad \Rightarrow \quad g \sqsubseteq_{\text{lw}} e, \qquad \textit{and} \tag{5.3}$$

$$h \sqsubseteq_{\text{pre}} i \wedge k \sqsubseteq_{\text{child}} h \wedge m \sqsubseteq_{\text{child}} i \quad \Rightarrow \quad k \sqsubseteq_{\text{lw}} m. \tag{5.4}$$

The order $\sqsubseteq_{\text{child}}$ determines the spacetree's structure. The order $\sqsubseteq_{\text{pre}}$ on all children of each refined element makes the traversal deterministic.

The distinction of the level-wise and a standard depth-first order results from the difference between (5.1) and (5.3). The latter makes the traversal visit all children of one node first before it descends further. A refined element's k^d children thus are available en bloc (as a patch) before the algorithm continues on the subsequent level (Algorithm 5.1). If the tree is processed sequentially, the traversal's uniqueness is recovered by (5.4), i.e. the traversal again is a total order, and—in the case of a space-filling curve defining $\sqsubseteq_{\text{pre}}$—follows the Peano space-filling curve. While the traversal still imposes a total order, I consequently weaken the order on the events and can first load all the data, then trigger the events in parallel, and finally store the elements.

The traversal modification deserves two additional remarks. On the one hand, the term level-wise implies that the traversal is a mixture of two traversal paradigms: The overall traversal remains of depth-first type. Within one set of children, the traversal though equals a breadth-first approach. On the other hand, the technique of recursion unrolling [65] also leads to a level-wise depth-first order. k^d pairs of push and pop operations on the call stack are replaced by one recursion step holding the data simultaneously, i.e. the traversal replaces two levels (one recursion step) with one code fragment. This code fragment then can employ more sophisticated

Algorithm 5.1 Level-wise depth-first traversal without traversal events and vertex handling. The algorithm starts with $lwDFS(e_0)$, and the enumeration of the children e_i obeys a given traversal order $\sqsubseteq_{\text{pre}}$.

$$\mathcal{T} \quad = \quad (\mathbb{E}_{\mathcal{T}}, \sqsubseteq_{\text{child}} \in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}, e_0 \in \mathbb{E}_{\mathcal{T}}, \mathbb{V}_{\mathcal{T}})$$

1: **procedure** $lwDFS(e)$
2: $\quad e_{\text{child}} \leftarrow (\bot, \bot, \ldots)$ ▷ $|e_{\text{child}}| = k^d$
3: $\quad$ **for** $i = 1 : k^d$ **do**
4: $\quad\quad e_{\text{child},i} \leftarrow e_i = \mathbf{pop}_{\text{input}}(), \qquad e_i \sqsubseteq_{\text{child}} e$ ▷ e_i stem from the input stack
5: $\quad$ **end for**
6: ▷ Trigger *enterElement* events, etc.
7: $\quad$ **for** $i = 1 : k^d$ **do**
8: $\quad\quad lwDFS(e_{\text{child},i})$
9: $\quad$ **end for**
10: ▷ Trigger *leaveElement* events, etc.
11: $\quad$ **for** $i = 1 : k^d$ **do**
12: $\quad\quad \mathbf{push}_{\text{output}}(e_{\text{child},i})$ ▷ Store children on output stack
13: $\quad$ **end for**
14: **end procedure**

algorithms, as a larger part of the grid—k^d elements and their vertices instead of one single element—is available. Interpreting the level-wise depth-first order as image of a recursion unrolling motivates that a stack-based grid management for $k = 3$ still works: the geometric elements are replaced by 3^d patches as atomic elements, and the 2^d vertices within these patches are directly transported from the input stream to the output stream. The stack sequence consequently alters, but the data access pattern for vertices on the patch boundaries remains unchanged.

5.1.1 k-spacetree Decomposition

Triggering up to k^d events in parallel is a poor level of concurrency. Yet, it is a good starting point for a more elaborate scheme. In the following, I at first introduce a colouring for the k-spacetree. Each colour later represents one computing node's responsibilities. While its definition states properties of the colouring, a formal construction is given later on. For the time being, I elaborate a parallel traversal scheme combining a given colouring and the level-wise depth-first traversal. This parallel scheme then in turn states the properties of an efficient colouring in Section 5.3.

Let p denote the number of computing nodes. Each node has a number: the rank.

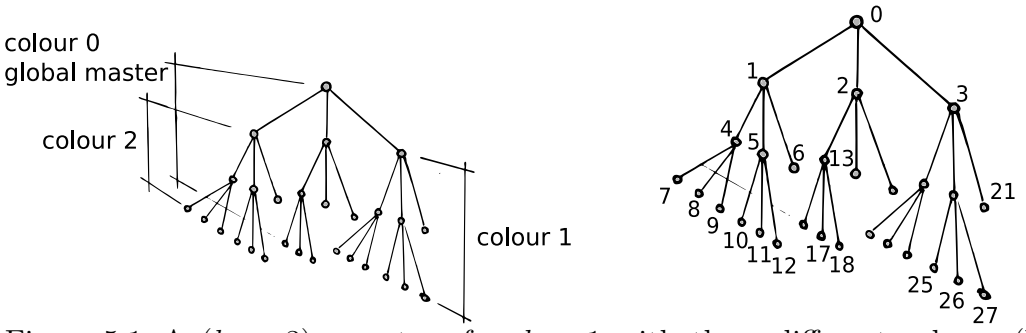


Figure 5.1: A ($k = 3$)-spacetree for $d = 1$ with three different colours (left). A level-wise depth-first enumeration of the same spacetree (right).

Assign each element of a spacetree $\mathcal{T}$ an attribute holding the rank number of the program instance responsible for this element. At the beginning, the attributes' values are undefined $\perp$. For each rank besides 0, choose an arbitrary spacetree element with marker $\perp$ besides the spacetree's root. Set it to the rank, and, afterwards, all unmarked descendants of the element are assigned this rank, too. Finally, all the remaining elements with $\perp$ are set to rank 0. It is at least the spacetree's root element.

The colouring introduces a tree topology on the involved computing nodes if each node handles one colour (Figure 5.1), and it defines a distributed adaptive grid hierarchy [61]: each node is responsible for a subtree of the overall k-spacetree, i.e. it holds a small k-spacetree itself. If subparts of this small k-spacetree are coloured with a different rank, the decomposition delegates work on these parts to further computing nodes. The tree topology thus introduces a master-worker relationship: Rank 0 is a global master not working for anyone. Each other rank is a worker for one master. Each rank delegating work also acts as master for other nodes.

5.1.2 Parallel Level-wise Depth-first Traversal

With a decomposition at hand, the recursive traversal becomes a parallel traversal (Algorithm 5.2, Figure 5.2): The global master starts the spacetree traversal. All the other nodes wait. On each processor exactly the same recursive program is executed—a *single program multiple data* paradigm. The traversal stack automaton loads k^d subelements in each recursion step. Before it performs operations on the subelements or steps down further, it analyses which subnodes are assigned to another remote rank. It first informs all the remote ranks to start their traversal. Then, it steps down into the nodes assigned to itself. As remote nodes handle elements assigned to them as well as their descendants, the local algorithm exclusively steps down into subtrees assigned to itself. As soon as all local subtrees have been processed, the algorithm waits for all remote nodes to finish. Afterwards, the

Algorithm 5.2 Extension of Algorithm 5.1 exploiting a given k-spacetree decomposition. The result is a parallel, level-wise depth-first traversal.

```
1:  procedure lwDFS(e)
2:      e_child ← (⊥, ⊥, ...),        |e_child| = k^d
3:      for i ∈ {1, k^d} do
4:          e_child,i ← e_i = pop_input(),        e_i ⊑_child e    ▷ e_i stem from the input stack
5:      end for
6:      ...                                                  ▷ Trigger enterElement events, etc.
7:      for i ∈ {1, k^d} do
8:          if e_i is remote then
9:              Startup remote k-spacetree with root e_i
10:         end if                                           ▷ All remote nodes are started up
11:     end for
12:     for i ∈ {1, k^d} do                                  ▷ Do local work on finer levels
13:         if e_i is not remote then
14:             lwDFS(e_child,i)
15:         end if
16:     end for
17:     for i ∈ {1, k^d} do                                  ▷ Local work is done
18:         if e_i is remote then
19:             Wait for remote k-spacetree's result from element e_i
20:         end if
21:     end for
22:     ...                                                  ▷ Trigger leaveElement events, etc.
23:     for i ∈ {1, k^d} do
24:         push_output(e_child,i)                           ▷ Store children on output stack
25:     end for
26: end procedure
```

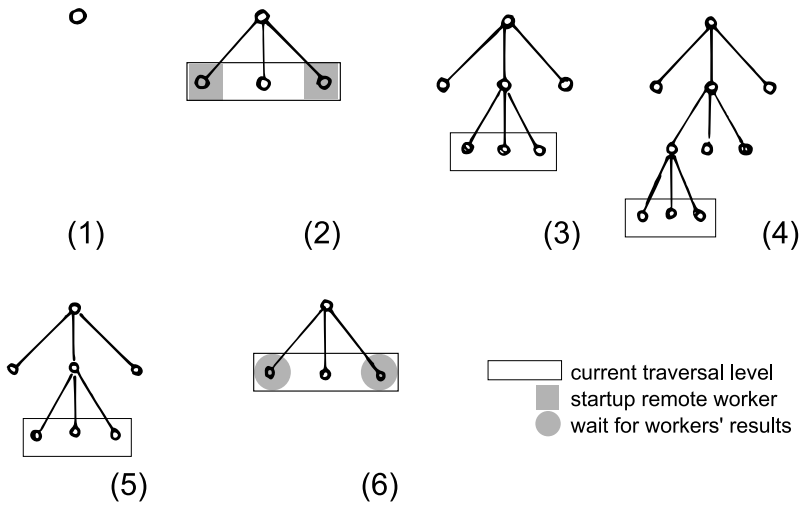


Figure 5.2: Parallel level-wise depth-first traversal: The traversal starts at the root element (1). Whenever a remote element is loaded, it starts up the traversal on the worker (2). Such remote elements are never refined locally. Then, it follows the level-wise depth-first order (3-4) and ascends again (5). If a remote element is passed throughout the ascend, the traversal waits for the remote node's results, i.e. for the remote traversal to finish (6).

algorithm continues to step up.

The description reveals a number of properties of the parallel traversal:

- At the beginning, only one node is active. All the other nodes wait for an activation.
- The deeper the global master descends in the k-spacetree, the more computing nodes become involved in the computation.
- The higher the global master ascends in the k-spacetree, the fewer computing nodes work.
- The individual nodes are synchronised due to their root nodes, i.e. as soon as a worker is activated, it runs completely independent of its master. Finally, the traversals' termination is synchronised as the master waits for its workers to finish.

The first three issues reveal a bottleneck: the startup phase, where the global master starts the traversal but has not activated its workers which in turn have to activate their workers. This phase corresponds to a difficulty all multiscale algorithms face: The finer the problem, the more work to distribute. In turn, it always happens that a problem becomes too coarse to exploit all computational resources.

The synchronisation discussion in the fourth issue reveals that information such as steering data (current smoother level, refinement policy, and so forth) and grid properties (number of spacetree elements, information whether solver has triggered a refinement, and so forth) are passed through the tree bottom-up and top-down—information analysis and inheritance [46]. Information inheritance is mirrored by recursion steps of the traversal automaton and startup calls for remote nodes. Information analysis corresponds to a call stack depth reduction, i.e. return statements within the recursive block or finish messages sent by the workers to their master, respectively.

5.1.3 Multiple Top-level Elements

So far, each worker handles one k-subspacetree, i.e. each worker's workload corresponds to a tree identified by one root element. Such an approach leads to scaling problems, since a master has to wait for all remote nodes whenever it leaves a k^d patch, and the following example as well as Figure 5.3 illustrate this.

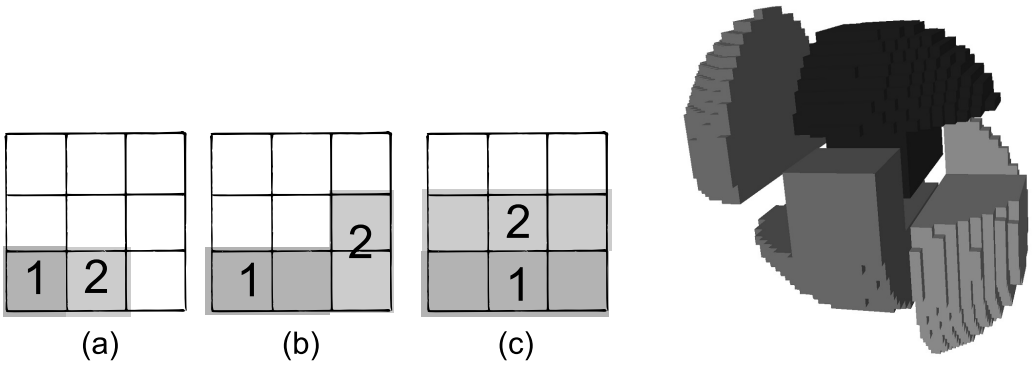


Figure 5.3: In a gedankenexperiment, a node has booked two workers 1 and 2 for a 3^d patch ($d = 2$). (a) If all children of the patch induce a spacetree of the same depth, and if each worker can handle only one child, the splitting into three processes results in a bad work balancing, as the workload of the initial process is reduced by two units, but the workload of the workers equals one unit. (b) If each worker handles two subtrees, the work balancing improves (see also Figure 5.4). (c) The best balancing corresponds to three subtrees per worker. Sphere domain split up among four nodes (right).

Example 5.2. *Let $\mathcal{T}$ be a spacetree yielding a regular Cartesian grid for $d = 2, k = 3$. The height of $\mathcal{T}$ is h. $p \in \{2, \ldots, 3^d\}$, and the nine children of the root element are distributed among the p nodes. Each remote tree has height $h - 1$, i.e. $\sum_{i=0}^{h-1} 9^i$*

elements. As $p-1$ elements on the first level are deployed to remote nodes, the global master has to descend into $(9-(p-1))\sum_{i=0}^{h-1} 9^i$ elements before it starts to receive the remote nodes' finish messages. The workload on the global master is $10-p$ times higher than the remote nodes' workload, i.e. the workload is not well-balanced if $p \neq 9$.

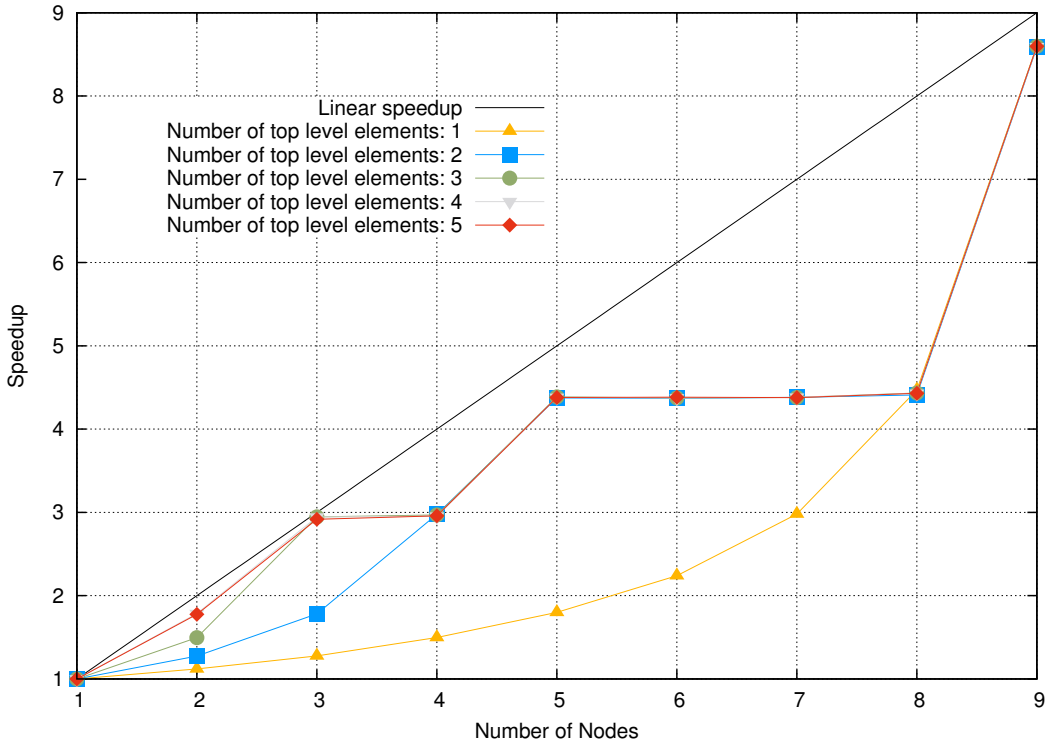


Figure 5.4: One k-spacetree per node results in a poor load balancing if the number of computing nodes does not fit to the grid. Here, the grid is regular ($d = 2$) with $6.09 \cdot 10^5$ vertices. The speedup improves with an increasing number of spacetrees (top level elements) mapped to one rank. Results stem from the Infinicluster.

The example's considerations lead to three important insights for regular grids:

1. The number of computing nodes has to fit to the grid to make a domain decomposition result in a good speedup, i.e. for $p = k^d$ the balancing works fine, but for $p < k^d$ no good balancing exists.

2. The bigger the spacetree becomes, the stronger the effect of the imbalance due to a growing height h in the example.

3. For $p > k^d$, the gedankenexperiment yields the insight that a good decomposition exists only for $p = \left(k^d\right)^i$. Consequently, the balancing problem worsens with an increasing number p of computing nodes, i.e. the number of nodes has to grow exponentially to make the traversal benefit from additional computing power.

If the algorithm is allowed to map several spacetrees to one rank, this problem can be weakened (Figure 5.4). Nevertheless, a step pattern for the speedup curve remains, since the rank with the maximum number of elements or subspacetrees, respectively, determines the traversal time.

Example 5.3. *Let $d = 2, k = 3$. I study an optimal colouring with four computing nodes, where the first node (master) has to distribute all nine children of a refined element among the four participants. Let the traversal furthermore scale linearly with the number of geometric elements processed. If the workload is split up equally, three out of four nodes handle two children. The remaining computing node is responsible for three children and determines the traversal time, if all children induce spacetrees of the same height. Having only three nodes at hand would lead to the same speedup, as the speedup correlates to the maximum number of elements per computing node.*

The implementation later recognises situations as sketched above, and it uses only a minimal number of nodes: Assuming that all children induce spacetrees of the same height, it gives back workers that would not improve the performance further.

5.2 Partitions Induced by Space-Filling Curves

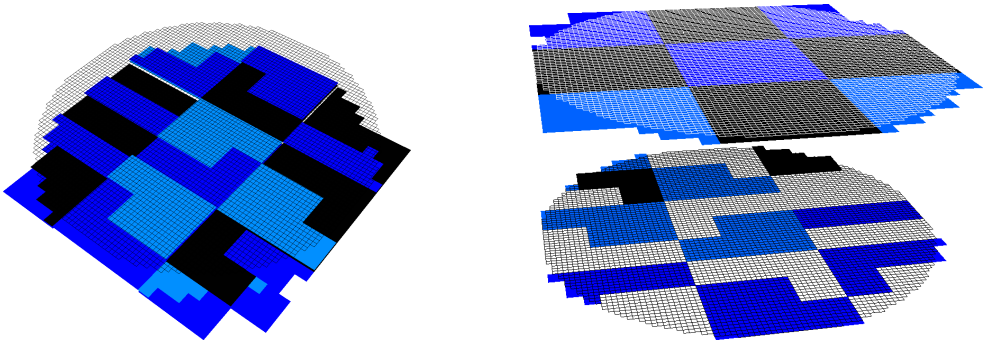


Figure 5.5: Two illustrations of a domain decomposition for circle domain. On the right-hand side, the underlying logical topology of the individual subdomains.

Peano's traversal runs through the individual spacetree levels along the Peano curve's iterates. Let the partitioning follow this traversal idea, too: If one node holds two spacetrees corresponding to two geometric elements e_1 and e_2, these elements should be siblings, and they should be neighbours along the Peano space-filling curve's iterate.

As a result, the fine grid partitions resemble subcurves of the Peano iterates (Figure 5.5), and the parallel performance benefits from this. Several factors influence the efficiency of a parallel algorithm: First, the whole amount of exchanged data influences the algorithm's performance, as it defines the bandwidth required for the communication. Second, the merge of received data with the local data along the partition's boundary influences the algorithm's performance, as it is realised by additional code. Third, the number and the size of individual messages influence the algorithm's performance, as the higher this number, the higher is the impact of the network's latency and the communication realisation[1]. Forth, the parallel performance relies on a good load balancing—perhaps the most important aspect. Finally, additional technical factors such as the logical topology, the load balancing overhead, and communication latency codetermine the actual speed. The following pages study the realisation aspects one to three, and I neglect the technical factors. Furthermore, I assume that the tree colouring has already introduced a perfect balancing.

5.2.1 Quasi-Optimal Partitions

The *bandwidth* determines the maximum amount of data per time a network can exchange. If the amount of data for a communication task is known, the bandwidth hence determines the time required by one node to send the data to another node as one big bunch. It is a common wisdom that clever parallel implementations make the exchange of data and the computations overlap: The nodes exchange data and, meanwhile, they continue to compute. The computation on each node in turn continues as long as its input does not depend on remote data not exchanged yet. If this dependency makes the computation wait, the algorithm is communication-restricted. This restriction becomes the more dominant the more data is exchanged. Otherwise, the complete communication is hidden from the runtime. Communication facilities in many supercomputers are extremely limited, and many PDE simulation codes run the risk of becoming bandwidth- or latency-restricted—if the underlying data sets are scaled up, the amount of exchanged data scales up, too. As a result, the scaling's influence on the computing time and the exchanged data is important.

For a given partitioning, each computing node handles one partition, i.e. the partition's volume determines the workload (amount of operations) of the node.

[1] Architectures such as the Infinicluster provide for example direct memory access mechanisms, as long as all messages fit into a hardware buffer.

At the same time, the individual ranks have to exchange information along the partitions' boundary. The bigger the volume is, the higher is the computational load. The bigger the surface is, the bigger the amount of exchanged data.

It is the easier to make the communication run in the background of the computation, the smaller the partition's surface compared to its volume. A good partitioning yields a big *volume-surface ratio* [17, 31] often referred as high *quality coefficient* of the partitions [42], high *partition locality* [28], or *surface-to-volume* if it is defined as minimisation problem [44].

Hyperspheres exhibit a minimal surface $s_{\min}$ with respect to their volume V given by their radius $r(V)$. A partition hence is optimal, if it converges to a hypersphere with decreasing mesh size.

$$\begin{aligned}
s_{\min}(r) &= r^{d-1} \cdot \frac{2\pi^{d/2}}{\Gamma(d/2)}, \qquad \text{and} \\
r(V) &= \sqrt[d]{V \cdot \frac{\Gamma(d/2+1)}{\pi^{d/2}}} = \frac{V^{1/d}}{\sqrt{\pi}} \left(\frac{d}{2} \cdot \Gamma(d/2) \right)^{1/d}, \qquad \text{i.e.} \\
s_{\min}(V) &= V^{\frac{d-1}{d}} \cdot C_{\text{optimal}}, \qquad \text{with} \\
C_{\text{optimal}} &= \frac{2\pi^{d/2}}{\Gamma(d/2)} \cdot \frac{1}{\pi^{\frac{d-1}{2}}} \cdot \left(\frac{d}{2} \Gamma(d/2) \right) \cdot \sqrt[d]{\frac{2}{d \cdot \Gamma(d/2)}} \\
&= \sqrt{\pi} \cdot d^{\frac{d-1}{d}} \sqrt[d]{\frac{2}{\Gamma(d/2)}}.
\end{aligned} \tag{5.5}$$

It is impossible to split up an arbitrary computational domain into disjoint hyperspheres. However, one can achieve a *quasi-optimal* partitioning yielding subdomains with a volume-surface ratio as good as the optimal partitioning besides a constant, i.e. (5.5) holds with a constant $C \geq C_{\text{optimal}}$. For meshes, the volume equals the number of geometric elements assigned to a partition, and the surface equals the number of boundary vertices.

Theorem 5.1. *The partitions induced by the Peano space-filling curve's iterate on a regular grid are* quasi-optimal.

The proof of the theorem in [17] is two-fold: First, it analyses the fine grid of a partition. Its surface is bounded by the surface of the bounding box plus the contributions of concave parts of the subdomain. The bounding box corresponds to a construction step of the iterate. The concave contributions correspond to subsequent construction steps, and, hence, they can be analysed by a recursive formulation. Second, the proof takes the fine grid surface's estimate and adds the vertices belonging to coarser grids of the k-spacetree. It ends up with

$$s(V) \leq V^{\frac{d-1}{d}} \cdot \frac{4d}{1-3^{1-d}} 3^{d-1}.$$

For the continuous case, i.e. partitions corresponding to an infinite small mesh width, the Hölder continuity of the space-filling curve [66] immediately yields an analogous estimate.

5.2.2 Parallel Peano Spacetrees

A node is allowed to hold several k-spacetrees. To avoid duplicated data and to work within a homogeneous algorithmic environment, these spacetrees are stored within one single data structure: Besides the spacetree's root elements, each node also holds their common parent. Yet, it does not perform any operation on this parent, as the parent belongs to the master's partition. Individual subspacetrees then integrate seamlessly into one data structure.

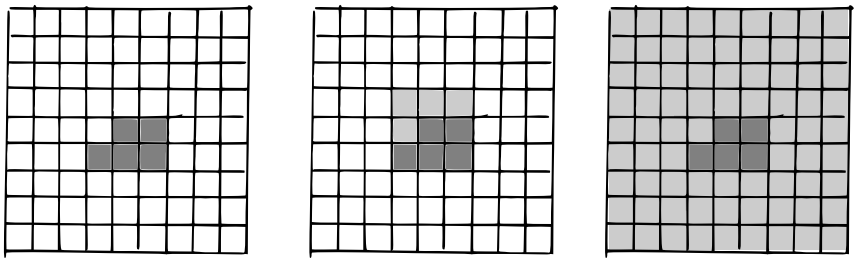


Figure 5.6: As soon as a partition is found (left), the algorithm augments the partition's spacetree. First, it adds elements to have a k-spacetree again (middle). Then, it embeds the whole spacetree into k^d elements (right).

The definition of a k-spacetree comprises the completeness of each level, i.e. each refined element holds k^d children. To make the integrated spacetree fit to this definition—this avoids a modification of the underlying traversal algorithm—the spacetree is augmented with the missing geometric elements and vertices. The additional records are set to outside, and, hence, the node does not perform any operation on them[2]. Such an augmentation is common for many approaches (see [54] and papers cited therein). The induced memory overhead is studied in a moment.

Boundary vertices in the Peano implementation are stored persistently according to Section 2.7. For the parallel realisation, an additional advantage of this convention becomes apparent: A k-spacetree representing a subtree also might have "boundary" vertices that are inside the domain, and, thus, persistent. Their attributes have to be available in each iteration. Hence, the parallel Peano spacetree is also embedded into a coarser grid. As a result, all parallel domain boundary vertices are held persistently on the input and output streams, if a stack-based traversal is chosen. Vertices stemming from the data structure augmentation are set to outer if they are not adjacent to any element assigned to the local rank.

[2]the inside/outside definitions in Section 2.7 have to be altered accordingly.

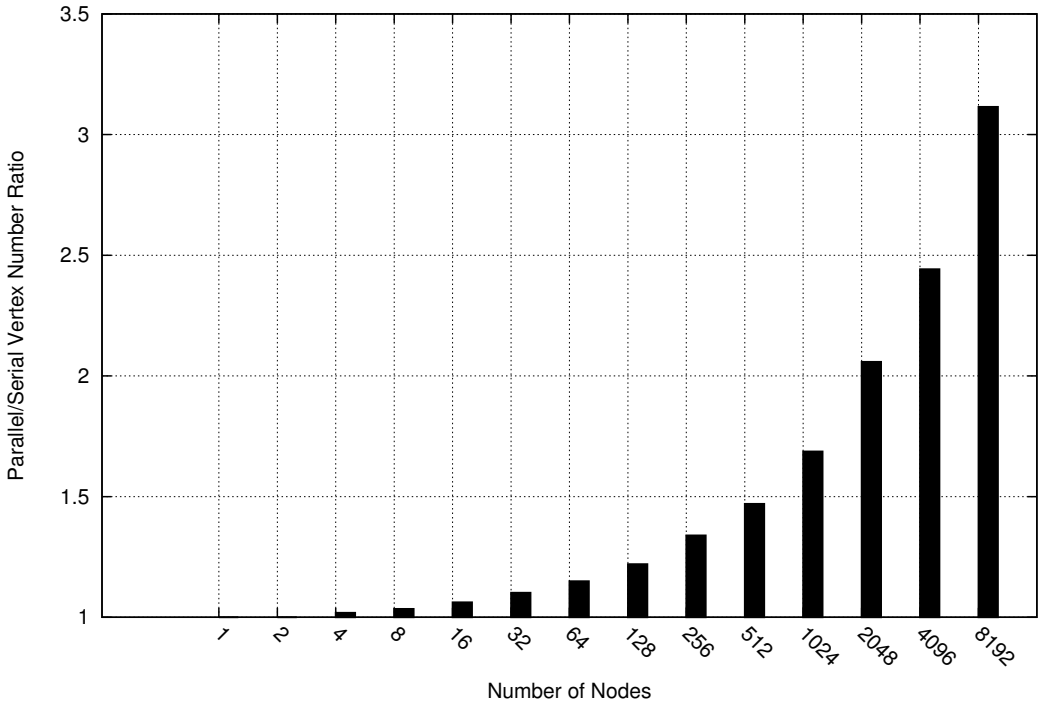


Figure 5.7: Memory overhead resulting from the two-step augmentation of the individual node's k-spacetrees. The figures result from a $d = 2$ experiment with a regular grid for a square domain and $6.09 \cdot 10^5$ vertices in the sequential code.

The two-step augmentation of the spacetree (Figure 5.6) entails a memory overhead. This overhead is bounded, as, first, the partition's boundary is a submanifold with a reduced dimension, and, second, the surrounding grid is chosen as coarse as possible—outer vertices are never refined. The measurements in Figure 5.7 give a worst case estimate of this overhead for a strong scaling, i.e. the problem size is fixed and the number of computing nodes is increased. Although the overhead appears to be impressive (more than a factor of three for around 8000 nodes), this overhead has to be broken down to the individual processes, i.e. the overhead per process equals the overall overhead divided by the number of ranks. In the following discussion, I hence neglect it.

5.2.3 Vertex Data Exchange

The domain decomposition has to exchange the boundaries of the individual partitions: If a vertex is adjacent to elements belonging to different nodes, this vertex

is held on each node locally. Some of its data such as refinement flags have to be exchanged to enable the individual subspacetrees to keep the grids and the corresponding data consistent. In the following, I use the terms vertex and the exchanged data as synonyms.

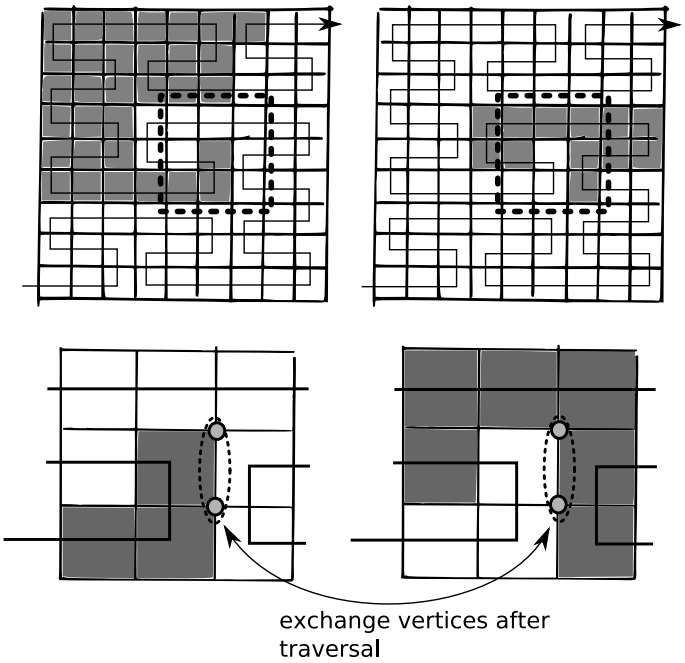


Figure 5.8: There is no need to resort the exchanged vertices: The k-spacetree augmentation and the space-filling curve ensure that the vertices are exchanged in the right order, i.e. the send order equals the output stream's order.

All actions of the parallel traversal split up into operations belonging to a computational phase followed by a communication phase comprising the remaining actions. The latter phase exchanges boundary data with all neighbouring nodes: Each vertex belonging to the subdomain's boundary is sent to all the nodes holding a copy of the vertex. Afterwards, the copy sent by all the communication partners is received and merged into the local copy. The computation phase works exclusively on the local copy. According to the idea of the communication running in the background of the computation, both communication and computation phase are merged. A vertex is sent away as soon as it is written to the output stream (*fire-and-forget semantics*). Its remote contributions have to be received when the vertex is again loaded from the input stream, but there is no need to have a vertex's data already available when the subsequent iteration starts.

With a given partitioning for two computing nodes, the vertex exchange pattern resembles an X-pattern: Both nodes traverse their subspacetree and send data to the

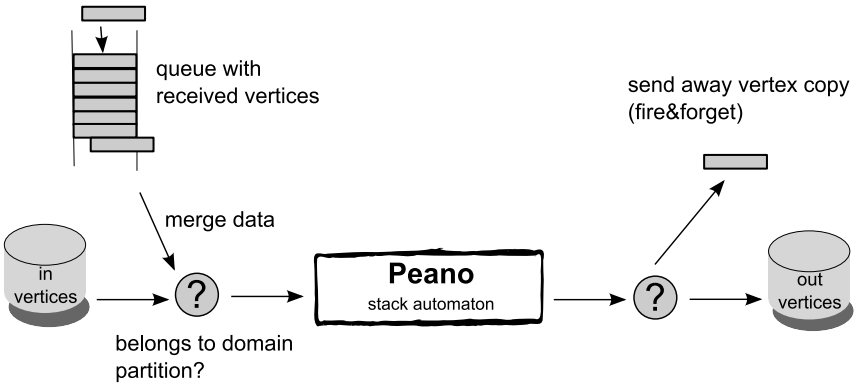


Figure 5.9: For each vertex loaded from the input stream, the algorithm checks whether this vertex belongs to the partition's domain. If this is the case, the vertex is the first vertex in the incoming vertex queue belonging to the communication partner. Throughout the vertex store process, copies are sent to the neighbours.

neighbour. The next traversal then combines the local vertex's state and the received data (Figure 5.8, 5.9, and 5.12). Such an exchange pattern imposes a restriction on the information speed: If a vertex's state is altered, this state transition is not available at remote nodes before the next iteration. This fits to the discussion in Section 2.6, where the refinement and coarsening are adopted to the information speed restriction. No additional effort thus is to be spent on the parallel data consistency management.

Each node receives data and has to merge the received information into the local vertex input stream. If, for example, a received vertex holds the flag `refinement-triggered`, this flag of the local vertex copy has to be set before it is passed to the grid traversal. A naive implementation of this merge either takes the received vertex and searches for the corresponding vertex in the input stream. Or it searches within the queue of received vertices as soon as a boundary vertex is to be loaded from the input stream. Without loss of generality, such a search has $\mathcal{O}(s \log s)$ complexity with s representing the partition boundary's size.

Here, this sorting overhead however disappears completely due to the space-filling curve and the k-spacetree augmentation (Figure 5.8 and Figure 5.9): All nodes traverse their k-spacetree simultaneously in accordance with the global Peano iterate. Although the individual iterates and the subdomains do not cover each other, the order of the boundary vertices on the output stream is globally unique: Each node has an input and output stream of its own. They are different on each node. However, if a boundary vertex a is stored to the output stream before a boundary vertex b is stored, this relation holds on all nodes holding both a and b.

Received messages are stored in one queue per communication partner. The queue

is treated as stack: At the end of an iteration, the queue waits until its size equals the number of sent messages. Afterwards, it deploys the messages starting with the last message received, i.e. whenever the traversal reads a vertex from the input stream, it analyses whether this vertex belongs to the parallel subdomain's boundary. If this is the case, there are $1 < n \leq 2^d - 1$ remote nodes holding this vertex, too. For each node, there is a receive queue. Their copy of the vertex is the message lying on top of this queue. Whenever the grid traversal writes a vertex to the output stream, the stack management analyses whether this vertex belongs to the parallel subdomain's boundary. If this is the case, there are $1 < n \leq 2^d - 1$ remote nodes waiting for a copy in the next iteration. The local node copies the vertex n times and sends all the copies to the remote nodes. Afterwards, the vertex is stored on the output stream.

A straightforward fire-and-forget realisation—the outgoing vertices are sent individually—often leads to performance breakdowns, as each send entails a certain overhead resulting from network latencies, e.g. To tackle this issue, the implementation uses an additional buffer per communication partner. The outgoing messages are stored within this buffer, and its content is sent away as one huge message as soon as the buffer is full or the traversal terminates. The receive process also receives the messages in blocks. As the receiving node does not read the received data before the subsequent iteration, the buffering strategy is hidden from the exchange process and does not impose any restriction or imply any modification of the exchange pattern.

5.2.4 Parallel Boundary Vertex Realisation

The preceding section defines the vertex exchange pattern. Yet, it lacks a concrete realisation, and, in particular, it does not define how to identify whether a vertex belongs to the partition's boundary, which ranks' partitions are adjacent, and how this adjacency information is altered whenever the colouring of the k-spacetree changes. Due to the element-wise traversal, the automaton holds only a small number of geometric elements at a time, and it can not derive adjacency information from the elements—particularly, elements being neighbours but not siblings are never available at the same time, i.e. a partition border in-between such elements can not be detected. The boundary information thus is stored within the vertices.

Domain Adjacency Lists

Each vertex holds two domain adjacency lists

$$\begin{aligned} thisLevel : \mathbb{V}_{\mathcal{T}} \times \{0, \ldots, 2^d - 1\} &\mapsto \mathbb{N}_0 \cup \{\perp\}, \qquad \text{and} \\ subLevel : \mathbb{V}_{\mathcal{T}} \times \{0, \ldots, 2^d - 1\} &\mapsto \mathbb{N}_0 \cup \{\perp\}. \end{aligned}$$

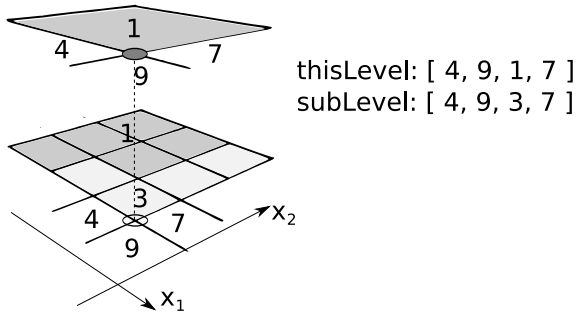


Figure 5.10: For each vertex (marked), *thisLevel* holds the ranks of the nodes responsible for the adjacent geometric elements (top). *subLevel* holds this information for the subsequent level (bottom).

thisLevel stores the rank of the nodes responsible for the 2^d elements that are adjacent to the vertex. Its entries are lexicographically enumerated. An entry $\perp$ denotes that the rank is unknown or unimportant. *subLevel* holds the same information for the vertex at the same position on the subsequent level: if a vertex becomes refined, *subLevel* gives the new vertex's *thisLevel* mapping. Consequently, both lists are related yet not equal if any adjacent element is refined and holds a child assigned to a different rank.

Let *rank* denote the number (rank) of a computing node processing a vertex. The following facts either derive from the semantics of the two mappings or fit naturally into the concept:

- $v \in \mathbb{H}_{\mathcal{T}}$ implies that the vertex does not hold information. Ergo, there is also no adjacency information stored: $\forall i \in \{0, \ldots, 2^d - 1\} : thisLevel(v, i) = \perp \wedge subLevel(v, i) = \perp$.
- $\forall i \in \{0, \ldots, 2^d - 1\} : thisLevel(v, i) = rank$ implies that the vertex v is located within the partition, i.e. it is surrounded by elements assigned to *rank*. It does thus not belong to the partition's boundary.
- $\exists i \in \{0, \ldots, 2^d - 1\} : thisLevel(v, i) \neq rank \wedge thisLevel(v, i) \neq \perp$ implies that the vertex v belongs to the partition boundary, i.e. whenever *rank* stores the vertex on the output stream, it has to send a copy to node $thisLevel(v, i)$. In turn, whenever the stack delivers the vertex, there's also a copy from node $thisLevel(i)$ available in the message queue.
- $\forall i \in \{0, \ldots, 2^d - 1\} : thisLevel(v, i) \neq rank$ implies that vertex v is not an element of *rank*'s domain. Thus, the algorithm converts the vertex's list into $thisLevel(v, i) \mapsto (\perp, \perp, \ldots)$, and $subLevel \mapsto (\perp, \perp, \ldots)$.
- $\forall i \in \{0, \ldots, 2^d - 1\} : thisLevel(v, i) \neq rank$ furthermore implies that $\neg \mathcal{P}_{\text{refined}}(v)$.

The invariant

$$\forall i \in \{0,\ldots,2^d-1\}: \quad thisLevel(v_i, 2^d-1-i) = rank \quad \vee \\ thisLevel(v_i, 2^d-1-i) = \perp \tag{5.6}$$

holds for all the 2^d non-hanging vertices of an element e processed by node $rank$. i here enumerates the element's vertices lexicographically. Whenever an element is deployed to a new worker $rank_{\text{worker}}$, all its entries update according to

$$\forall i \in \{0,\ldots,2^d-1\}: subLevel(v_i, 2^d-1-i) \leftarrow rank_{\text{worker}},$$

and the vertices of the finer levels follow the inheritance Algorithm 5.3. For vertices inside a partition, *thisLevel* and *subLevel* have the same content.

Vertex Merge

The definition of the adjacency lists finally allows to formalise the merge process (Algorithm 5.5). The merge of local vertices and data from remote nodes consists of three steps:

1. If a neighbouring node has booked a new worker, or if the neighbour has merged with its master, the local node has to update the local adjacency lists, as the tree colouring has changed. Otherwise, the local node would send the vertex to the old, invalid neighbouring process at the end of the iteration. Entries of the adjacency list may differ from the local entries if and only if they hold the remote node's rank, i.e. each node is responsible for "its" adjacency entries. If entries in the receive queue differ from the local entries, the local node updates its vertices.

2. If the remote vertex has triggered a refinement, the refinement has to be triggered on the local node, too. As a result, the state `refinement-triggered` is set, and the grid structure's consistency is preserved. For the coarsening, the same arguing holds.

3. Finally, the merge process has to merge the PDE-specific data. Chapter 5.5 for example elaborates this part of the merge process for the Poisson multigrid algorithm.

With the realisation details written down, Peano can handle distributed spacetrees. The boundary exchange and data synchronisation fit smoothly into the stack concept proposed in Chapter 3, and it is clear that Peano's spacetrees represent quasi-optimal partitions if the workload is balanced. Balancing thus is the missing link addressed by the upcoming section.

Algorithm 5.3 A new vertex's adjacency information derives from the coarser level's vertices (Figure 5.10). *vertices* holds the coarse element's vertices, and *coordinates* defines the new vertex's position within a k^d patch. The operation relies on a recursive operation with the same name and an extended signature.

$$\begin{aligned} derive : \mathbb{V}_{\mathcal{T}}^{2^d} \times \{0,\ldots,k-1\}^d \times \mathbb{V}_{\mathcal{T}} &\mapsto \mathbb{V}_{\mathcal{T}} \\ derive : \mathbb{V}_{\mathcal{T}}^{2^d} \times \{0,\ldots,k-1\}^d \times \mathbb{V}_{\mathcal{T}} \times \{-1,\ldots,d-1\} &\mapsto (\mathbb{N}_0 \cup \{\perp\})^{2^d} \end{aligned}$$

```
1: procedure derive(vertices, coordinates, result)
2:     thisLevel(result) ← derive(vertices, coordinates, result, d − 1)
3:     subLevel(result) ← derive(vertices, coordinates, result, d − 1)
4: end procedure
5: procedure derive(vertices, coordinates, result, dim)
6:     if dim = −1 then
7:         coarseCoord ← (0, 0, ...)
8:         for all i ∈ {0, d − 1} ∧ coordinates_i = k − 1 do
9:             coarseCoord_i ← i
10:        end for
11:        return subList(vertices_i)
12:    end if
13:    if coordinates_dim = 0 ∨ coordinates_dim = k − 1 then
14:        return derive(vertices, coordinates, result, dim − 1)
15:    end if
16:    smallCoord ← coordinates ∧ bigCoord ← coordinates
17:    smallCoord_dim ← 0 ∧ bigCoord_dim ← k − 1
18:    smallList ← derive(vertices, smallCoord, result, dim − 1)
19:    bigList ← derive(vertices, bigCoord, result, dim − 1)
20:    return mergeTwoLists(smallList, bigList, dim)        ▷ see Algorithm 5.4
21: end procedure
```

Algorithm 5.4 Helper for Algorithm 5.3.

$$\begin{aligned} mergeTwoLists : \mathcal{A} \times \mathcal{A} \times \{0,\ldots,d-1\} &\mapsto (\mathbb{N}_0 \cup \bot)^{2^d} \\ \mathcal{A} := \{0,\ldots,2^d-1\} &\mapsto \mathbb{N}_0 \cup \{\bot\} \\ i &\in \mathbb{N}_0^d \end{aligned}$$

```
1: procedure mergeTwoLists(smaller, bigger, axis)
2:     result ← (⊥, ⊥, ...))
3:     for all i ∈ {(0,0,...),(1,1,...)} ∧ i_axis = 0 do
4:         iOpposed ← i
5:         iOpposed_axis ← 1
6:         tmp ← smaller_iOpposed
7:         if smaller_iOpposed ≠ bigger_i then    ▷ Invariant (5.6) for coarser element
8:             if smaller_iOpposed = ⊥ then
9:                 tmp ← bigger_i
10:            end if
11:            if bigger_i = ⊥ then
12:                tmp ← smaller_iOpposed
13:            end if
14:        end if
15:        result_i ← tmp ∧ result_iOpposed ← tmp
16:    end for
17:    return result
18: end procedure
```

5.3 Work Partitioning and Load Balancing

The preceding pages take a colouring fixing the partitions of the k-spacetree, make the partitions fit again to the k-spacetree idea, introduce the concept of cuts along the Peano space-filling curve, and discuss the realisation of the vertex exchange. Yet, they do not argue how to determine the tree decomposition. This section derives a splitting algorithm that fits to the level-wise depth-first traversal and balances the workload among the computing nodes.

Classical load balancing algorithms focus on a homogeneous load distribution, i.e. they try to assign all nodes of a parallel machine the same workload: Then, no node is a bottleneck and the parallel speedup is maximised. The approach here is twofold and adds an additional reasoning: On the one hand, it balances the workload. On the other hand, external factors such as the grid structure, the load distribution overhead, and so forth often determine how many nodes can be employed economically, i.e. using more nodes does not improve the performance. In

Algorithm 5.5 If a remote vertex and a local vertex are merged, the adjacency lists and the refinement flags have to be updated. Thereby, each node is allowed to modify its own partition.

$$mergeWithNeighbour : \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} \times \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}} \times \mathbb{N}_0 \quad \mapsto \quad \mathbb{V}_{\mathcal{T}} \setminus \mathbb{H}_{\mathcal{T}}$$

```
1: procedure mergeWithNeighbour(local, neighbour, rankOfNeighbour)
2:     for all i ∈ {0, 2^d − 1} do
3:         if thisLevel(i) = rankOfNeighbour then
4:             thisLevel(local, i) ← thisLevel(neighbour, i)
5:             subLevel(local, i) ← subLevel(neighbour, i)
6:         end if
7:     end for
8:     if ¬P_refined(local) ∧ P_refinement triggered(neighbour) then
9:         P_refinement triggered(local)
10:    end if
11:    if P_refined(local) ∧ P_coarsening triggered(neighbour) then
12:        P_coarsening triggered(local)
13:    end if
14:    return local
15: end procedure
```

such a case, the concept here tries to use as few nodes as possible leaving the remaining nodes idle. The rationale is that the grid changes throughout the computation, i.e. the load balancing has to be redone permanently. Having idle nodes at hand simplifies this rebalancing.

The non-functional properties of the algorithms are twofold, too: As the grid changes permanently, the algorithm implements an on-the-fly load balancing, i.e. the grid is permanently rebalanced. Alternative algorithms stop the computation from time to time and redistribute the complete workload. Consequently, they require a heuristic or a rule-of-thumb when it is worth to stop the simulation. My approach gets along without stopping the computation. Next, as the traversal and multigrid algorithm never set up global data structures, the distribution algorithm also refrains from a global data structure. Instead, the whole load balancing is integrated into the element-wise traversal, and the decisions are made locally, i.e. the load balancing runs in parallel, too.

The balancing's design follows a greedy paradigm:

- It assumes that a good k-spacetree decomposition is already available for a given, static k-spacetree. If the spacetree consists of only one element, such a decomposition is trivial, as it employs only one node.

- It assumes that (enough) idle nodes are available.
- Whenever an algorithm on a node triggers a refinement somewhere in the grid, the algorithm analyses whether this refinement introduces a new global bottleneck, i.e. whether the refinement slows down the traversal. If this is the case, it takes an idle node and adds it as new worker to the refining node. As a result, the bottleneck is eliminated a priori, before the grid is actually refined.
- Whenever a master-worker relationship in the tree decomposition could be removed without making the master slow down the overall traversal, the master and the worker a merged. Thus, the merged worker becomes idle and is available for further decompositions.

5.3.1 Weight Attribute

The balancing requires a cost model. Before a redistribution concept is elaborated, the algorithm has to be able to measure the time spent on the traversal. It is a straightforward idea to assign each geometric element a weight representing an abstract cost function. This weight mirrors the element-wise traversal and a linear time assumption: If the number of geometric elements is doubled, the time per traversal doubles, too. This linear cost model in turn is validated by the (almost) constant time per vertex in Section 3.6.

Definition 5.2. *The* weight *of a geometric element represents the time the traversal spends in this element and all the descendants of the element.*

Let $w : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{N}_0^+$ denote the weight with

$$C_{\text{weight}} := 1, \tag{5.7}$$

$$w(e) = C_{\text{weight}} \quad \forall e \in \mathbb{E}_{\mathcal{T}}, e \text{ is a leaf, i.e. } \neg\mathcal{P}_{\text{refined}}(e). \tag{5.8}$$

$$w(p) = \begin{cases} C_{\text{weight}} + w_{\text{remote}}(p) & \text{if } \mathcal{P}_{\text{wait}}(p) \\ C_{\text{weight}} + w_{\text{local}}(p) & \text{if } \neg\mathcal{P}_{\text{wait}}(p) \end{cases}, \tag{5.9}$$

$$w_{\text{remote}}(p) = \max_{c \sqsubseteq_{child} p,\ c \text{ remote}} \{w(c)\}, \tag{5.10}$$

$$w_{\text{local}}(p) = \sum_{c \sqsubseteq_{child} p,\ c \text{ not remote}} w(c), \quad \text{and} \tag{5.11}$$

$$\mathcal{P}_{\text{wait}}(p) = w_{\text{remote}}(p) \geq w_{\text{local}}(p). \tag{5.12}$$

The weight model is homogeneous, i.e. the computational cost per geometric element are constant (5.7). If this assumption is not valid anymore, the constant C_{weight} has to be replaced by a function modelling the computational load. A leaf's weight is prescribed (5.8), whereas a refined element's weight results from the computational load of the element itself and all the descendants (5.9): First, let w_{local} denote the

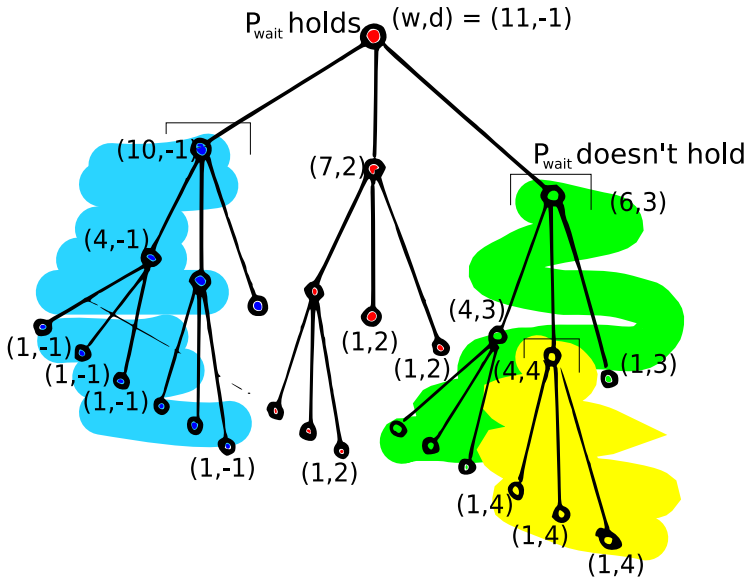


Figure 5.11: The weight and the δ attribute on a k-spacetree split up into four parts.

time spent in all the descendant (5.11). Second, w denotes for each remote node the weight, i.e. the time the worker spends in this element[3]. As all the workers' traversals run in parallel, the remote time equals the maximum of the remote runtimes (5.10). Third, a node either needs longer than all the remote nodes to process its own subelements, or it has to wait for a remote node. In the latter case, predicate $\mathcal{P}_{\text{wait}}$ in (5.12) holds. Finally, the weight of a refined element with remote subelements either is dominated by the workers' runtime ($\mathcal{P}_{\text{wait}}$ holds) or by the local traversal ($\mathcal{P}_{\text{wait}}$ does not hold). Whenever a node is responsible for all the children of a refined element, the element's weight w equals the sum of the runtime spent within the children added one cost unit for the node itself (5.11). The attribute w can be evaluated within the traversal's bottom-up steps. It is synthesised.

5.3.2 δ Attribute

Load balancing is a discrete optimisation problem. This thesis follows a local greedy algorithm: it improves the load distribution on each node locally, since it evaluates for each node permanently whether it is a bottleneck or whether it is allowed to do more work without becoming a bottleneck. The weight attribute gives the computational load per subtree, while the wait predicate identifies bottlenecks: If the wait predicate holds, a worker thwarts its master. Yet, the weight lacks the information how much additional work could be added without slowing down the algorithm—it

[3]If several remote spacetrees are assigned to one node, w has to be modified accordingly.

supports a bottleneck analysis, but does not identify bottlenecks in advance. If an element refines, additional work is added, and it is important to know whether this additional work would slow down the application.

Definition 5.3. *The attribute δ for a geometric element identifies how many additional work units can be spent on the element and all its successors without making the corresponding traversal a bottleneck.*

Let $\delta : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{Z}$ give a runtime difference for each node. It defines how many additional cost units (geometric elements) could be spent on the element and its successors without making the worker responsible for the element a bottleneck for its master. If $\delta < 0$, the worker is already a bottleneck. If $\delta = n > 0$ holds, the worker will not become a bottleneck, if up to n elements are added to the subtree identified by the node. δ is an inherited attribute. It is computed throughout the top–down steps and uses the weight attribute w of the preceding traversal:

$$\delta(e_0) = -1, \qquad \text{for the spacetree's root, and} \tag{5.13}$$

$$\forall c, p \in \mathbb{E}_{\mathcal{T}},\ c \sqsubseteq_{\text{child}} p : \quad \delta(c) = \begin{cases} \delta(p) + w(p) - w(c) - 1 & c \text{ and } p \text{ processed on different nodes} \\ \delta(p) + w(p) - w(c) - 1 & \text{same node } \wedge\, \mathcal{P}_{\text{wait}}(p) \\ \delta(p) & \text{same node } \wedge\, \neg\mathcal{P}_{\text{wait}}(p). \end{cases}$$

Attribute δ tells the algorithm where a fork would improve the overall runtime of the traversal, as the root node is assigned a negative δ in (5.13). Whenever a child and its parent are handled on the same node, and the parent does not have to wait for another remote node, the child has the same δ as the parent: If the parent may spend n additional work units on the grid, the child is also allowed to do so—the parent is already refined and can not get additional work. If the parent is too slow by n work units and slows down the whole application, the traversal on the subtree has to speed up. The parent itself can not speed up, as it has k^d children to be processed. Thus, it delegates the speed up requirement to its children. Whenever a child and its parent are handled on the same node, and the parent has to wait for another remote node, the child node can gain weight without slowing down the traversal. Whenever a child and its parent are handled on different nodes, the child's node is a worker of the parent's node. In both of the latter cases, the difference between the parent's weight and the child's weight equals the additional cost the traversal could spend on the child. The decrement eliminates the weight of the parent node from the δ calculation. A combination of both attributes now enables each node to decide recursively whether it would or does benefit from additional workers handling subtrees. This strategy is elaborated in the following two sections.

5.3.3 Fork Predicate and the Node Pool

My implementation makes use of a worker pool. At the beginning, all the computing nodes except the initial one processing the spacetree's root are idle workers assigned to the worker pool. Whenever a computing node wants to delegate work, it tries to book a worker from the worker pool.

If a leaf $l, l \sqsubseteq_{\text{child}} p$, is to be refined due to a `refinement-triggered` flag and $\delta(l) = n \leq k^d$ holds, the refinement would slow down the overall computation. Throughout the descends, the traversal then determines in element p how many of p's leaves would refine. Afterwards, it tries to book up to $k^d - 1$ workers from the node pool in order to distribute those leaves among the new workers along the Peano iterate. If the worker pool is empty, the computing node has to do the work for the new leaf itself.

Example 5.4. *Let $d = 3, k = 2$. Six out of eight children of a refined element want to refine, and the refined element's δ-attribute is negative. The refined element is already a bottleneck of the overall computation, and the additional refinements would worsen this bottleneck. The responsible node thus tries to book five additional workers—deploying all refined elements to workers would make the node holding the refined element an idle node, as it has to wait for its workers. Instead of five workers, the node pool delivers only three workers. The first worker is assigned the first two refining leaves along the Peano iterate, the second worker is assigned the leaves three and four along the Peano iterate, the third worker is assigned the fifth element, and the master itself handles the remaining sixth refined element.*

Since the worker deploys exclusively the refining elements along the Peano curve, the resulting remote partitions may be disconnected: If the non-refining elements are refined later throughout the computation, for example due to an a posteriori error estimator, the domain decomposition ends up with disconnected fine grid partitions. While one can avoid this by deploying solely connected refined elements, I prefer a balanced to a connected partitioning in the experiments.

Definition 5.4. *The fork predicate for a spacetree leaf holds, if a refinement of the leaf would make the corresponding node a bottleneck.*

The predicate is given as

$$\forall e \in \mathbb{E}_{\mathcal{T}} : \ \mathcal{P}_{\text{fork}}(e) = \begin{cases} \top & \text{if } \neg\mathcal{P}_{\text{refined}}(e) \wedge \delta(e) \leq k^d \\ \bot & \text{else} \end{cases} .$$

As it depends only on δ, the algorithm computes the predicate on-demand.

While an application's performance typically motivates the parallelisation, many applications also suffer from hard memory restrictions. In such a case, the fork predicate should also evaluate an upper memory bound, i.e. if the process would

exceed a given memory threshold, the fork predicate holds, too. Such an augmented fork predicate makes the load distribution fit to the memory available on the individual nodes, i.e. a node never exceeds the main memory due to a refinement, and it improves the applicability and stability for long-running, real-world simulations. A similar argumentation though is found seldom in literature (yet [43], e.g., picks it up).

5.3.4 Join Predicate

The fork predicate makes the parallel traversal as fast as possible. In addition, the balancing shall use as few nodes as possible. Such a property is no value of its own: On the one hand, it reduces the exchanged data per iteration. On the other hand, it ensures that the node pool always holds as many idle nodes as possible. The following pages elaborate when nodes can be freed without harming the parallel performance.

Let there be a join predicate: If a node p has delegated the work of a child e and, thus, a complete subtree to a worker, it is the master of this worker. If $\mathcal{P}_{\text{join}}(p, e)$ holds, it is a *lazy master*, i.e. it could take over the job of the worker without becoming a bottleneck for its master itself.

$$\forall p, c \in \mathbb{E}_{\mathcal{T}}, c \sqsubseteq_{\text{child}} p : \mathcal{P}_{\text{join}}(p, c) = \begin{cases} \top & p, c \text{ processed on different nodes} \wedge \\ & \mathcal{P}_{\text{wait}} \wedge \\ & w_{\text{local}}(p) + w(c) < w(p) + \delta(p) \\ \top & p, c \text{ processed on different nodes} \wedge \\ & \neg\mathcal{P}_{\text{wait}}(p) \wedge \\ & w_{\text{local}}(p) + w(c) < w(p) - w(c) + \delta(p) \\ \bot & \text{else.} \end{cases}$$

Definition 5.5. *A* lazy master *is a node that delegates work to workers but could do this work without becoming a bottleneck. For the root nodes of workers working for a lazy master, the* join predicate *holds.*

As $\mathcal{P}_{\text{join}}$ depends on the tree attributes w and δ, the algorithm computes it on-demand. w and δ are always kept up-to-date. Whenever a node is refined, the fork condition is evaluated, and, thus, new workers might come into play. Whenever the join predicate holds for a node, the two corresponding spacetrees are merged, and the decruited computing node is reassigned to the worker pool.

If a k-spacetree's data is moved from a node to another node, it is always only from worker to master. This simple data movement paradigm preserves the tree topology on the computing nodes, and only complete subtrees are exchanged. The algorithm here does not allot the rebalancing of work packages smaller than a complete subtree.

While the join predicate evaluates the runtime behaviour, memory also can affect join decisions in several ways: It is not reasonable to join a worker with its master, if

the master would run out of memory because of this join. A join predicate evaluating the memory requirements besides the runtime behaviour is the counterpart of a fork predicate splitting up processes because of limited memory. The size of the worker also determines the amount of data required to join two partitions. As subdomains have to be moved from the worker to the master, a join can slow down the overall iteration due to bandwidth restrictions. In such a case, it is reasonable to restrict the number of joins to a fixed, small number—either globally or per master.

Although the fork and join predicate balance the spacetree according to the analytical model, the per-iteration evaluation of the predicates can introduce lots of forks and joins that finally prove to be unnecessary. For a less aggressive balancing, the experiments on the one hand make the join predicate evaluate the refinement state of the worker: If a worker refines any of its leaves, it is never merged into its master's partition. As a result, joins occur exclusively for invariant subtrees. On the other hand, the join predicates' left hand side is increased by a fixed constant, i.e. the join is performed if and only if the join would not introduce a bottleneck, even if the subdomain were bigger by this constant. The penalty constant mirrors the overhead introduced by a join and prohibits some joins that would become unfortunate because of the worker's tree changing later throughout the computation.

Although the domain splittings follow the Peano space-filling curve in this thesis, partitions assigned to one computing node can be disconnected. On page 182, I explain this fact with the interplay of the fork process and a dynamic refinement criterion. The merge process can also introduce disconnected partitions, as the join predicate does not check if the worker's and the master's fine grid partitions are connected. While one can incorporate such a check into the join predicate, I prefer an aggressive join behaviour to have as many idle workers as possible and, hence, accept disconnected partitions.

5.3.5 Join Process

The vertex merge rules in Section 5.2.4 exhibit how the vertices' adjacency lists and refinement flags are updated whenever the grid structure or the spacetree decomposition change. Yet, the rules do not work for partition joins: If a master and a worker partition join, their boundary vertices are given the new (master's) adjacency information. At the same time, the neighbours though send vertices to the merging worker.

To resolve this issue, the join is a two-step process (Figure 5.12). In the first traversal, the adjacency information of the worker's partition boundary is updated, and the algorithm sends away the updated vertices. In the second traversal, the worker receives the neighbour's data, merges it into the local spacetree, and, finally, sends its records to its master. Meanwhile, the neighbours update their adjacency lists and send their data to the master instead of the merged worker.

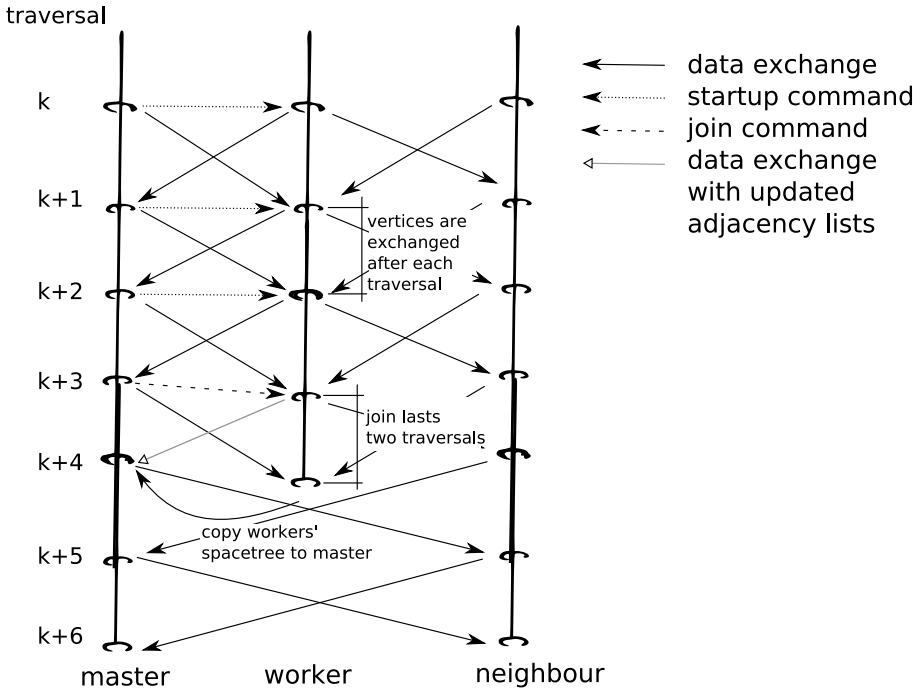


Figure 5.12: Vertex exchange pattern. Each join lasts two iterations.

5.4 Node Pool Realisation

A decentral load balancing as proposed in the preceding chapter has many advantages: As there is no central load balancing instance, this instance can not become a bottleneck. Furthermore, the load balancing exhibits exactly the overall algorithm's scaling. The only precondition is that the node pool answers requests for workers immediately, i.e. the node pool may not slow down the application. Thus, my implementation deploys the node pool administration to a process of its own.

Yet, the worker booking process remains a competition among the individual nodes, and the greedy fork often leads to unbalanced partitions[4]: Whenever the traversal comes across n refining leaves introducing a bottleneck, it tries books $n \leq k^d - 1$ workers. It does not make sense to book k^d workers, as the traversal has to wait for the workers throughout the ascends anyway. Thus, the node handles at least one refined element himself. Since the final grid layout is not known a priori—it might in fact change all the time—the algorithm does not yield an optimal partition, but permanently rebalances the partitions and, without loss of generality, never finds the optimal partitioning. This drawback holds for any greedy algorithm. Yet, an additional problem arises from the approach: If the node pool processes the worker requests on a first-come first-served (FCFS) basis, nodes with a slow network

[4]With number of nodes going to infinity, this drawback disappears, as each request can be fulfilled.

connection or a worker request that occurs late throughout the traversal are inferior.

A more sophisticated node pool with a *fair answering strategy* thus waits,

- until a certain time has elapsed (timeout),
- until each working node has posted a request, or
- until no further idle nodes are available.

It then permutes the worker requests and favours nodes that haven't booked workers before. In my implementation, a simple history holds the sequence of preceding requests, and the node pool sorts the request queue accordingly.

The timeout ensures that all nodes have a chance to post their requests, i.e. the race among the individual nodes is softened. Yet, the node pool starts to answer immediately, if the request queue's size equals the number of working nodes, or if there are no idle nodes left. In the latter case, it does not make sense to make any node wait, since the request will be answered negatively anyway. If the queue's size equals p, all the workers already wait for a request, i.e. no additional requests are on the way.

Besides the tailoring of the worker delivery, the strategy also allows to implement architecture-specific knowledge (also suggested in [43]): As the node pool finally decides whether a fork happens or not—it can always tell a node that there are no idle workers left—it can prohibit new master-worker relationships introducing runtime drawbacks. Furthermore, the node pool decides which new worker to deliver if there are several workers idle. Knowledge about the concrete hardware topology and the global logical topology typically influence both aspects.

5.5 Parallel Iterations and HTMG

The parallel algorithm exchanges the partitions' boundaries after each traversal. For the multigrid solver, this communication pattern entails some modifications. While the element-wise evaluation remains unaltered, results of the individual stencils are not always available throughout *touchVertexLastTime* events: Boundary vertices are adjacent to $k < 2^d$ (local) geometric elements. The contributions of the $2^d - k$ additional elements is available not until the remote vertices have been received, i.e. they are available throughout the $touchVertexFirstTime$ event of the subsequent traversal.

The parallelisation thus entails the following algorithm updates:

1. Temporary results such as the residual become persistent attributes. They are not discarded after the traversal.

2. Each accumulated value is exchanged. The vertex exchange hence comprises for example the residual in addition to the vertex refinement structure.

3. For each accumulated value, the boundary vertex's remote contributions are added to the local representation as soon as the vertex is read from the input stream. The boundary vertex merge process from page 170 hence is augmented by the accumulation statements.

4. In the parallel code, operations former triggered by $touchVertexLastTime$ are triggered by $touchVertexFirstTime$, as the underlying data is not available before the merge process ensures that all elements' contributions have been added to a vertex.

The enumeration reveals that the changes for the parallel solver are straightforward. Nevertheless, the latter issue also reveals that the parallel solver needs half an iteration more for a Jacobi step than the sequential code: The sequential code starts the Jacobi step throughout the first read (it clears the residual), and finishes the Jacobi step throughout the last write (it updates the value). The parallel code needs an additional $touchVertexFirstTime$ to update the vertex's value.

As the additional traversal results from the boundary exchange pattern, this arguing holds for all the other multigrid ingredients such as restriction and the subsequent coarse grid smoothing, too. The events are "shifted" on the time scale (Table 5.1) and arise "delayed" compared to the sequential code. Furthermore, the solver needs an additional state `firstSmoothingStepOnCoarserGrid`. In this state, it completes the restriction started in the preceding iteration besides the first smoothing step on the coarser grid. Because of the modified event mapping, the horizontal tree cuts also are delayed by one iteration: After a multigrid ascend, the traversal has to access the former active level $\ell_{\text{active}} + 1$ once more to restrict the right-hand side, i.e. the tree cuts may not yet store these values on a backup stream yet.

5.6 Experiments

The following experiments analyse the parallelisation and load balancing with three test setups elaborating algorithmic properties. They refrain from a real-world use case and have artificial character, and they do not apply any arithmetics, i.e. the Peano instances create the grid, traverse this grid, and trigger events. These events however are—besides the geometry analysis—mapped to empty operations, i.e. they deteriorate.

First, I study the impact of the message size on the overall performance, i.e. I determine appropriate, hardware-specific parameters for the subsequent experiments, and, thus, reduce the architecture's influence on the experimental results. Second, I study the parallel speedup for regular grids. Any parallelisation scheme has to

Table 5.1: Interplay of the traversal events, the solver states and the multigrid operations for the parallel code. The rule set employs a simultaneous coarse grid smoothing. `FirstCoarse` abbreviates `FirstSmoothingStepOnCoarserGrid`.

Solver State	Traversal Event	Description and Operations
`Smooth`	$createTemporaryVertex(v)$	$level(v) \leq \ell_{\text{active}} \Rightarrow$ interpolate coarse grid value.
	$touchVertexFirstTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ apply Jacobi update step and $r_v \leftarrow 0$ (clear residual) afterwards.
		$level(v) < \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step and $r_v \leftarrow 0$ (clear residual) afterwards.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil.
		$level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{unrefined } v}(e) \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	no operation.
`Ascend`	$createTemporaryVertex(v)$	$level(v) = \ell_{\text{active}} \Rightarrow v \leftarrow 0$, and
		$level(v) < \ell_{\text{active}} \Rightarrow$ interpolate coarse grid value.
	$touchVertexFirstTime(v)$	$level(v) = \ell_{\text{active}} \Rightarrow$ apply Jacobi update step, clear residual, and compute hierarchical transform.
		$level(v) < \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step and $r_v \leftarrow 0$ (clear residual) afterwards.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil.
		$level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{unrefined } v}(e) \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	no operation.
`FirstCoarse`	$createTemporaryVertex(v)$	$level(v) \leq \ell_{\text{active}} \Rightarrow$ interpolate coarse grid value.
	$touchVertexFirstTime(v)$	$level(v) = \ell_{\text{active}} + 1 \Rightarrow$ restrict $\hat{r}_v$, with $\hat{r}_v$ stored in variable r_v.
		$level(v) = \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step and $r_v \leftarrow 0$ (clear residual) afterwards.
		$level(v) = \ell_{\text{active}} \wedge \mathcal{P}_{\text{refined}}(v) \Rightarrow b_v \leftarrow 0$.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil.
		$level(e) < \ell_{\text{active}} \wedge \mathcal{P}_{\text{unrefined } v}(e) \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	no operation.
`Descend`	$createTemporaryVertex(v)$	$level(v) \leq \ell_{\text{active}} \Rightarrow$ interpolate coarse grid value.
	$touchVertexFirstTime(v)$	$level(v) < \ell_{\text{active}} \wedge \neg\mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step and $r_v \leftarrow 0$ (clear residual) afterwards.
		$level(v) = \ell_{\text{active}} - 1 \wedge \mathcal{P}_{\text{refined}}(v) \Rightarrow$ apply Jacobi update step and $r_v \leftarrow 0$ (clear residual) afterwards.
		$level(v) = \ell_{\text{active}} \Rightarrow$ compute inverse hierarchical transform.
	$enterElement(e)$	$level(e) = \ell_{\text{active}} \Rightarrow$ apply stencil.
	$touchVertexLastTime(v)$	no operation.

yield a satisfying parallel performance for such a setup, before it switches to more complicated experiments. Third, I study the parallel speedup for an adaptive grid resolving a singularity. To parallelise such a grid is challenging, as the underlying k-spacetree is extremely unbalanced, i.e. it is difficult for a domain decomposition to result in a parallel speedup at all. Furthermore, the grid construction process entails a permanent rebalancing, as the decomposition is not aware of the singularity in advance.

5.6.1 Message Size

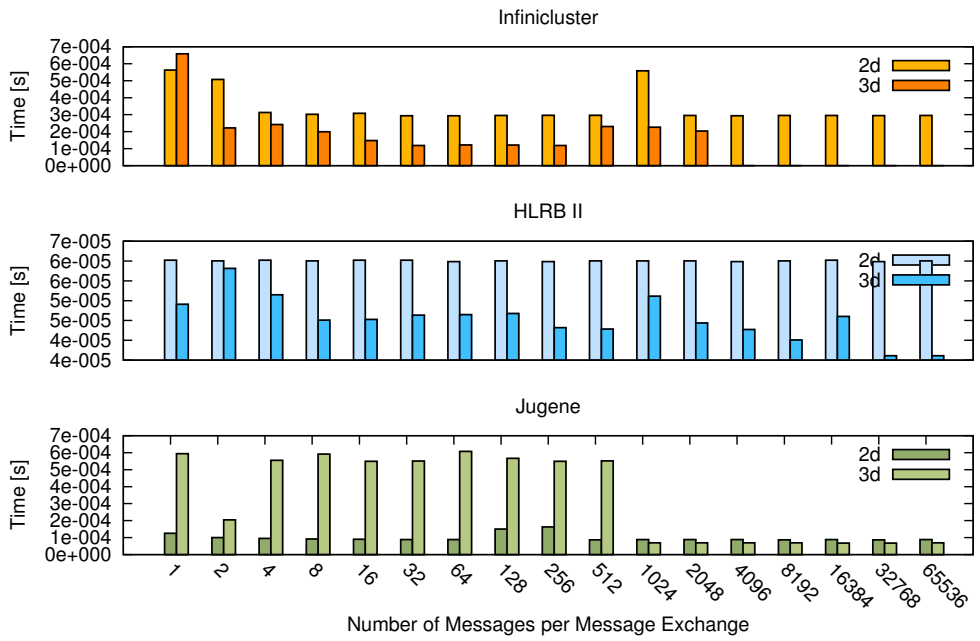


Figure 5.13: Influence of the message size on the runtime per vertex for three different architectures.

According to Section 5.2.3, several vertices are bundled into one big message bundle throughout the partition boundary exchange. This bundle's size depends on the attributes exchanged per vertex, on the bundle's cardinality, and on the data alignment. Albeit I study the correlation of size and performance, this section neither derives a global optimum size nor does it give a reasoning for the runtime behaviour. It just shows that such a performance study is critical—it anyway has to be done for each PDE individually—and derives reasonable settings for the grid records in the subsequent experiments.

For the measurements in Figure 5.13, a regular Cartesian grid is divided into equally sized subpartitions for k^d nodes. The parallelisation paradigm incorporates a hidden communication, since the vertex exchange follows a fire-and-forget semantics, i.e. the actual data exchange runs in the background of the traversal. Consequently, the runtime per vertex should be independent of the bundle's size. Nevertheless, the results reveal that the message size can influence the application's performance.

On the Infinicluster, a sufficiently big number of messages per bundle has to be chosen. Otherwise, the message exchange slows down the application (see one or two messages per bundle, e.g.). This effect is either a result of the network's latency, or it results from the overhead required to set up a data exchange. Choosing a too big message size for $d = 3$ also reduces the application's speed. As the effect does not occur for $d = 2$, and as the "three-dimensional" messages are bigger than the messages in two dimensions due to the cardinality of the adjacency lists depending on d, this is either a bandwidth restriction, or the Infinicluster's hardware fails to exchange very big messages in the background. I have no explanation for the peak for $d = 2$ and 1024 messages per bundle.

On the HLRB II, the results are two-faced: The performance is independent of the message size for $d = 2$. Yet, it is very sensitive to the cardinality for $d = 3$. This effect requires for further attention, but it is beyond the scope of this thesis. On Jugene, a BlueGene/P system, it is important to have a sufficiently big number of messages per bundle. The latter architecture is furthermore known to be very sensitive to data alignment [37], but the alignment's influence is not captured by the experiments at all.

5.6.2 Regular Cartesian Grids

A regular Cartesian grid corresponds to a balanced k-spacetree, i.e. each refined spacetree element of one level has the same number of descendants. Because of the simple, regular structure, the load balancing's behaviour can be reconstructed manually: The k^d elements on the level one are distributed among the cluster, since the root element holds $\delta = -1$ (the experiments of Figure 5.4 study the speedup for less than k^d computing nodes). Each of the k^d level-one elements holds $\delta = -1$, too, i.e. they try to re-fork immediately. If the number of remaining nodes is a multiple of k^d or bigger than $\left(k^d\right)^d$, each subtree forks the same number of times. Otherwise, the subtree with the smallest number of forks determines the overall runtime (Table 5.2).

Each of the k^d nodes responsible for the level-one elements tries to book additional workers. Because of this greedy approach, they compete for additional workers, and a first come first served (FCFS) node pool strategy might introduce a load imbalance non-deterministically, since it favours nodes with the best connection to the node pool. The fair strategy in Section 5.4 tackles this challenge.

Table 5.2: Forks accompany the grid creation for a regular grid, $d = 2$, different numbers of computing nodes, and one dedicated node pool (rank 0) implementing a fair answering strategy. Bold ranks are the global bottlenecks.

Nodes	Iteration	Vertices	Forks (master $\mapsto$ {masters and workers})
9	0	$6.80 \cdot 10^1$	$0 \mapsto \{\mathbf{2}, \mathbf{4}, \mathbf{1}, \mathbf{8}, \mathbf{3}, \mathbf{9}, \mathbf{5}, \mathbf{7}, \mathbf{6}\}$
	$\vdots$		
	7	$4.87 \cdot 10^7$	
17	0	$6.80 \cdot 10^1$	$0 \mapsto \{\mathbf{8}, \mathbf{4}, \mathbf{2}, \mathbf{16}, \mathbf{1}, \mathbf{13}, \mathbf{9}, \mathbf{12}, \mathbf{5}\}$
	1	$6.44 \cdot 10^2$	$1 \mapsto \{1, 17\}$, $2 \mapsto \{2, 6\}$, $4 \mapsto \{4, 15\}$, $5 \mapsto \{5, 3\}$, $8 \mapsto \{8, 14\}$, $9 \mapsto \{9, 10\}$, $12 \mapsto \{12, 11\}$, $13 \mapsto \{13, 7\}$, $16 \mapsto \{\mathbf{16}\}$
	$\vdots$		
	7	$4.89 \cdot 10^7$	
19	0	$6.80 \cdot 10^1$	$0 \mapsto \{\mathbf{4}, \mathbf{16}, \mathbf{2}, \mathbf{1}, \mathbf{8}, \mathbf{3}, \mathbf{6}, \mathbf{5}, \mathbf{12}\}$
	1	$6.44 \cdot 10^2$	$1 \mapsto \{1, 17, 11\}$, $2 \mapsto \{2, \mathbf{15}\}$, $3 \mapsto \{3, \mathbf{14}\}$, $4 \mapsto \{4, \mathbf{10}\}$, $5 \mapsto \{5, \mathbf{19}\}$, $6 \mapsto \{6, \mathbf{7}\}$, $8 \mapsto \{8, \mathbf{9}\}$, $12 \mapsto \{12, \mathbf{13}\}$, $16 \mapsto \{16, \mathbf{18}\}$
	$\vdots$		
	7	$4.89 \cdot 10^7$	
24	0	$6.80 \cdot 10^1$	$0 \mapsto \{\mathbf{16}, \mathbf{1}, \mathbf{4}, \mathbf{9}, \mathbf{8}, \mathbf{2}, \mathbf{5}, \mathbf{17}, \mathbf{7}\}$
	1	$6.44 \cdot 10^2$	$1 \mapsto \{1, 23, 6\}$, $2 \mapsto \{2, 21, 12\}$, $4 \mapsto \{4, \mathbf{3}\}$, $5 \mapsto \{5, 11, 10\}$, $7 \mapsto \{7, 15, 22\}$, $8 \mapsto \{8, 13, 14\}$, $9 \mapsto \{9, \mathbf{18}\}$, $16 \mapsto \{16, \mathbf{20}\}$, $17 \mapsto \{17, \mathbf{19}\}$
	$\vdots$		
	7	$4.90 \cdot 10^7$	

The experiments for the regular grid were at first conducted on the Infinicluster (Figure 5.14). For bigger node cardinalities, I switched to the HLRB II (Figure 5.15). All measurements illustrate a weak speedup—the grid size, i.e. spacetree depth, is scaled with the number of computing nodes—starting with $6.09 \cdot 10^4$ ($d = 2$) or $6.72 \cdot 10^5$ ($d = 3$), respectively, vertices on a single node, and both the HLRB II and the Jugene exhibited a similar speedup pattern besides some measurement inaccuracies.

The fork analysis above explains the step layout in Figure 5.14 as it applies recursively to all grid levels. This step pattern resembles the steps in Figure 5.4 scaled up to a bigger number of nodes. Furthermore, it becomes obviously that the fair node pool strategy delivers a more robust parallel decomposition compared to a FCFS implementation. I use the fair strategy throughout subsequent experiments.

The bigger the number of computing nodes and the bigger the spatial dimension d, the more additional nodes are needed to reach the next speedup level (Figure

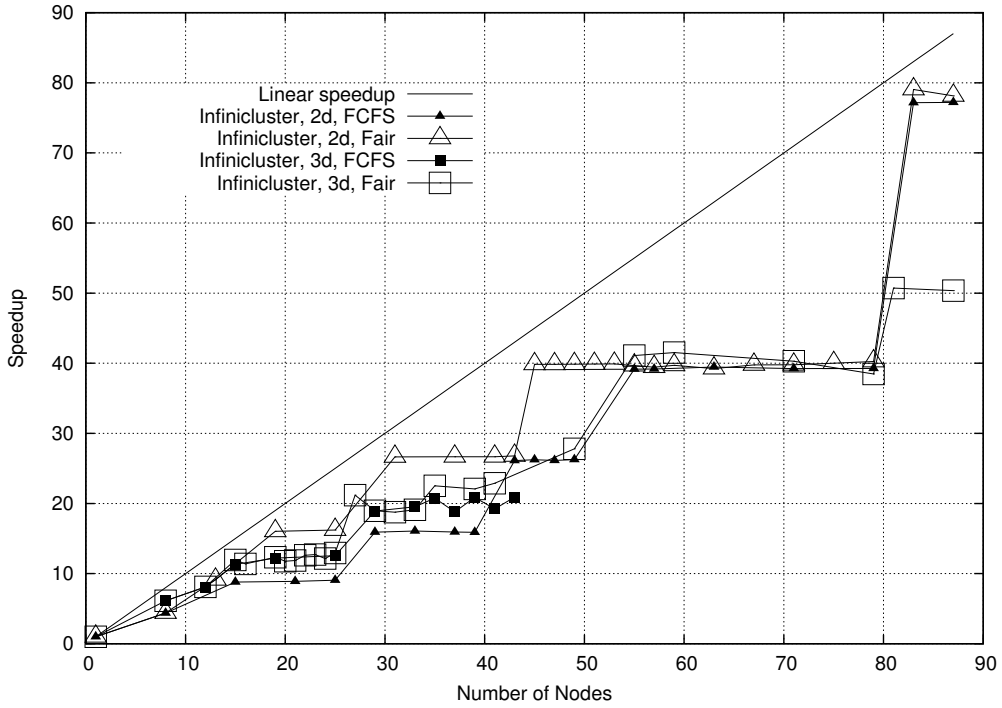


Figure 5.14: Weak speedup for a regular Cartesian grid on the Infinicluster.

5.15), since the work is distributed top-down and the number of nodes within the spacetree grows exponentially with $k^{id}, i \in \mathbb{N}_0$. If the number of computing nodes fits to the tree structure, the speedup for the regular grid is already promising. For the other cases, three remarks are important.

First, few applications yield that regular meshes due to more complicated boundaries or due to a more complicated solution behaviour. If one extracts the finest regular grid from such a mesh's k-spacetree, this grid's elements will hold non-uniform weights: Elements covering regions with a finer resolution have a greater weight than elements covering regions with a rather coarse mesh. Furthermore, Elements distributing work to additional nodes will have a smaller weight than elements doing all the work alone. The actual speedup curve pattern then adopts to the non-uniform weight distribution. Consequently, each application scenario will exhibit a different step pattern. A fair runtime comparison consequently has to average the speedup curves for different comparable scenarios to yield a reasonable speedup curve and to eliminate the influence of the nondeterministic worker assignment. The worst case steps from the measurement here then disappear, while the best case speedup results become worse.

Second, the measurements here apply a one-to-one mapping of Peano instances to computing nodes. In the outlook a modification and extension of this mapping is

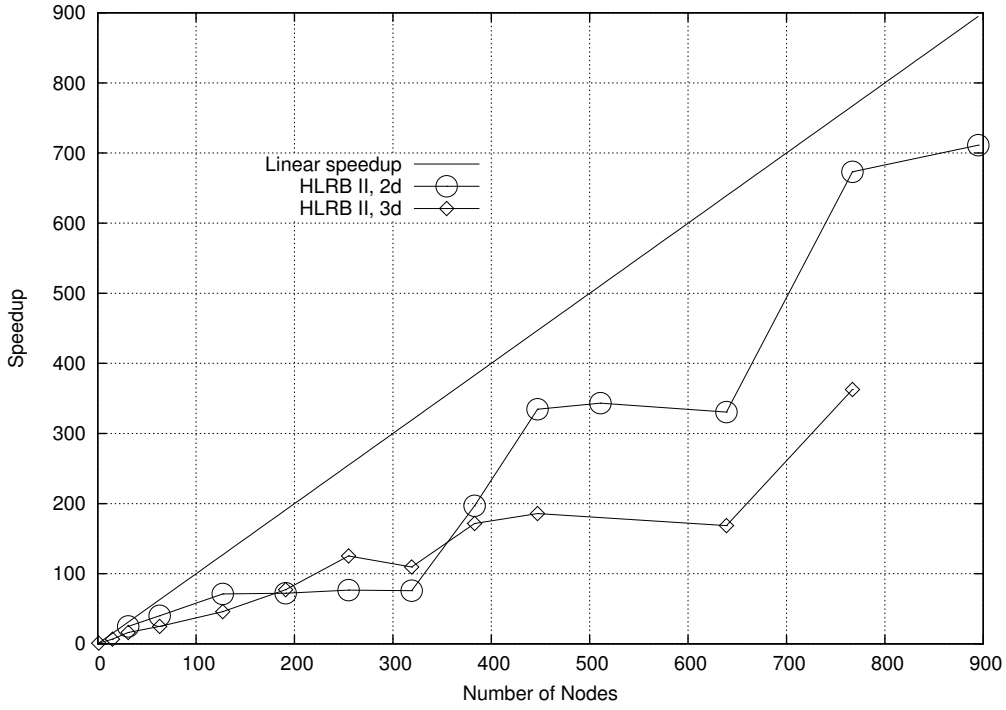


Figure 5.15: Weak speedup for a regular Cartesian grid on the HLRB II. The speedups of several runs are averaged, and the node pool implements a fair load distribution strategy.

discussed that should smooth out the speedup curve presented in the figures here.

Finally, the experiments employ Peano without a PDE solver. The computational work thus is exclusively determined by the grid management operations. If a PDE is solved, the additional floating point operations codetermine the runtime, and the poor slope for the three-dimensional problem should improve. If this is not the case, $d = 3$ has to be paid further attention in the future.

5.6.3 Singularities

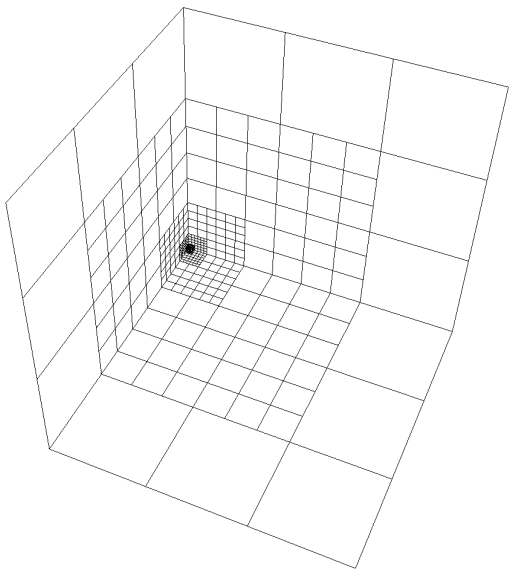


Figure 5.16: Grid resolving a singularity.

The last experiment resolves a point singularity at $x_0 = (1, 1, \ldots)^T \in \mathbb{R}^d$ on the unit square or cube, respectively: In each k-spacetree level, exclusively the vertices of the geometric element holding x_0 are refined (Figure 5.16). An optimal Peano traversal's runtime for such a spacetree scales with h being the spacetree's height, while the sequential runtime is $\approx h \cdot 2^d \cdot k^d$: the traversal has to step down into the finest elements once, while it can, on each level, deploy work for elements adjacent to the one element covering x_0 to remote nodes. If the runtime scales linearly in the number of elements to be traversed, they will always finish before their master starts to ascend again.

Again, it is straightforward to reconstruct the load balancing's behaviour: All elements covering x_0 have $\delta = -1$, i.e. their fork predicate holds. With each refinement, the corresponding node books additional workers. Two different patterns of load distributions then can occur: If the node deploys all elements besides the

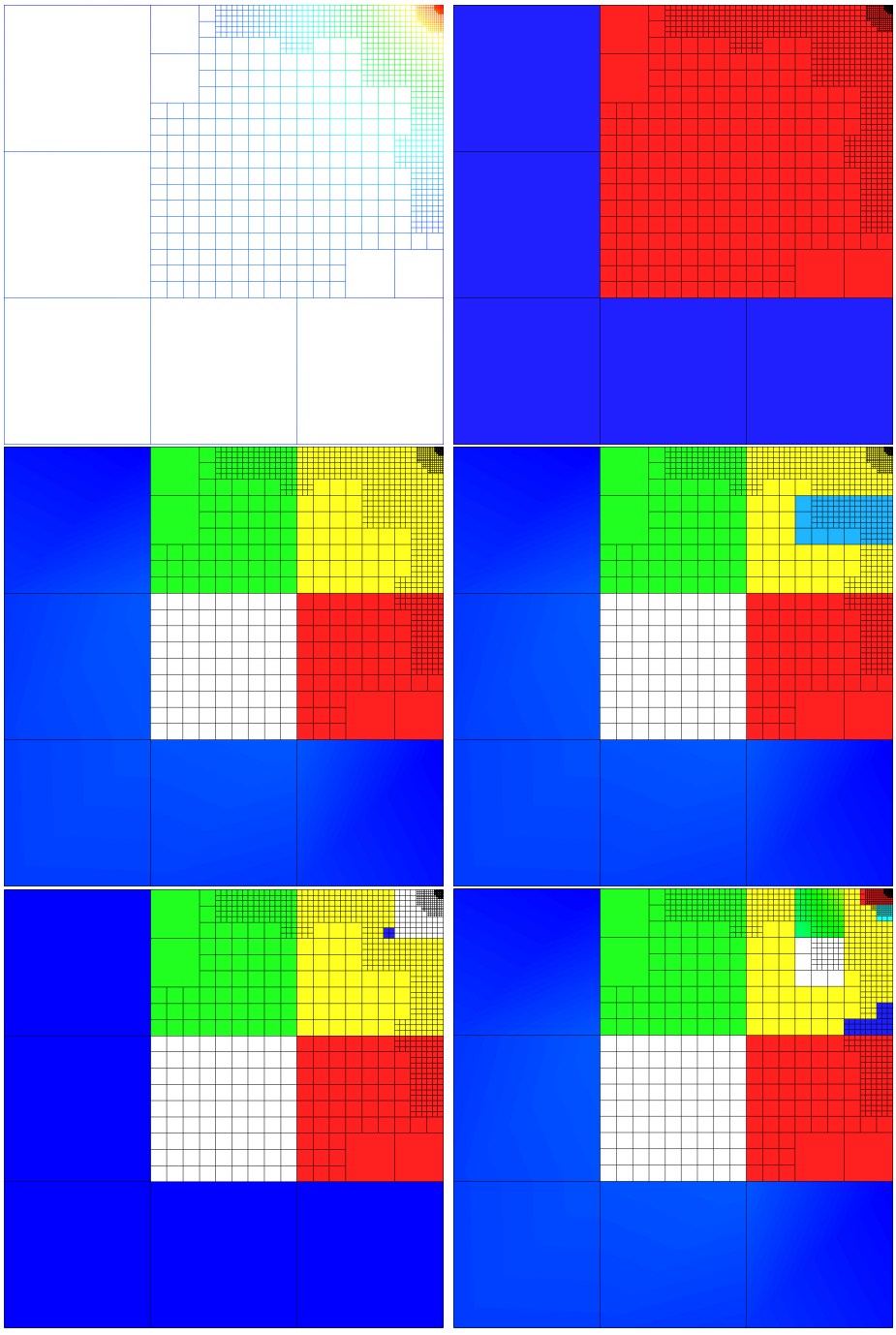


Figure 5.17: Grid resolving a singularity (top, left). The global master tries to deploy all work for refined elements to remote nodes, while it administrates the node pool itself (top, right). Below, in lexicographical order: Domain decompositions for $5, 6, 7$ and 10 nodes.

Table 5.3: Forks and merges accompany the grid construction in Figure 5.16 ($d = 2$). The rank holding the fine grid element covering x_0 is marked bold, and the transitions denote forks and joins.

Iteration	Vertices	Idle Nodes	Forks/Joins
0	$6.80 \cdot 10^1$	8	$0 \mapsto \{0, 6, 8, 5, \mathbf{3}\}$
1	$2.37 \cdot 10^2$	5	$\mathbf{3} \mapsto \{3, \mathbf{4}, 12, 10\}$
2	$4.06 \cdot 10^2$	2	$\mathbf{4} \mapsto \{\mathbf{4}, 1, 11, 9\}$
3	$5.75 \cdot 10^2$	1	$\mathbf{4} \mapsto \{4, \mathbf{7}\}$
4	$6.79 \cdot 10^2$	0	$\mathbf{7} \mapsto \{\mathbf{7}, 2\}$
5	$7.83 \cdot 10^2$	0	
6	$8.47 \cdot 10^2$	1	$\{3, 10\} \mapsto 3$
7	$8.86 \cdot 10^2$	0	$\mathbf{7} \mapsto \{7, \mathbf{10}\}$
8	$9.90 \cdot 10^2$	1	$\{3, 12\} \mapsto 3$
9	$1.03 \cdot 10^3$	0	$\mathbf{10} \mapsto \{10, \mathbf{12}\}$
10	$1.13 \cdot 10^3$	0	

element covering x_0 to workers, it remains the overall bottleneck: Throughout the steps down, it triggers the workers to startup their traversal. As they are refined only once, they always have finished their work before the master returns to step up. In a second case, the node deploys the element covering x_0 to a worker w_0. Then, w_0 becomes a bottleneck throughout the subsequent grid refinements, and the master will become a lazy master—it could take the workload of all workers besides w_0 without becoming a global bottleneck. Therefore, it triggers joins, and the freed workers are reemployed on a finer grid.

The corresponding experiments were conducted on the Infinicluster and showed a asymptotic maximum runtime improvement of $2^d \cdot k^d$ if the number of computing nodes is sufficiently large. Although the fork and join behaviour are nondeterministic, the structure of all operation traces is always the same. One example is given in Table 5.3. It agrees with the δ analysis above.

If Peano runs out of idle nodes throughout the grid construction, the runtime improvement breaks down (Figure 5.17 illustrates several examples): The application distributes the spacetree further and further until no nodes are idle anymore. The node being the global bottleneck is also responsible for the element covering x_0. This element refines further. As the node is already the slowest participant in the cluster, it never becomes a lazy master. No further joins are triggered by this lazy master, and the overall tree partitioning does not change its layout anymore.

An unbalanced tree such as the spacetree in Figure 5.16 is the worst case for any tree traversal algorithm. Many PDE solvers never yield such discretisations—our group's computational fluid dynamics code for example always delivers rather regular grids—besides for complicated domains. Having such a behaviour, the prac-

tical use of the experiments above is rather low regarding the runtime spent on the PDE's inner domain. It is however important that Peano books additional workers for grid singularities aggressively. In combination with speedup curves for regular grids (Section 5.6.2) this observation states that Peano's load balancing tackles regions with an adaptive, irregular structure aggressively with many computing nodes, and, hence, it diminishes runtime defects due to these regions.

5.7 Outlook

This closing section addresses some improvements of the parallelisation, while it leaves the fact aside that the implementation obviously needs additional attention, profiling, and tuning for $d \geq 3$. First, parameter studies and a subsequent tuning of the algorithm's magic constants within the predicate definitions adopt the parallelisation to a concrete hardware or a concrete PDE. The parallelisation yields nice speedups whenever the grid structure fits to both the traversal concept, the hardware topology, and the number of nodes available. To make the parallelisation robust with respect to these factors is, second, an important and outstanding challenge. Finally, the parallelisation concept here provides a vast set of links for methodological improvements.

Among the most important magic constants in the algorithm are the message queue size, the threshold constants in the fork and join predicate definitions, and the weight function. In Figure 5.13, the impact of the actual message size on the application's performance is illustrated. An optimal message size depends on both the cluster's hardware and the PDE's degree of freedom model, as the number of unknowns per vertex codetermines the size of one message and, thus, the number of bytes to be spent on a block of messages. It is laborious to determine a fitting message block size for each PDE model individually. Runtime models incorporating the network's latency, bandwidth, memory alignment, and so forth can select a message block size automatically and disburden the user from the laborious studies.

The constants in the predicate definitions also depend on the hardware used. Triggering a remote traversal for example entails a communication overhead. As a result, it sometimes is reasonable to merge two partitions although the master is not a lazy master. If the times spent on the additional local workload falls below the communication overhead, the overall runtime nevertheless improves, and an additional free worker becomes available. Such rationale, decisions, and experiences can be modeled by additional magic constants within the definition of the fork and join predicates.

k-spacetree leaves have a uniform cost model in this thesis, as their weight is fixed. For complicated PDEs, this concept has one shortcoming. If sophisticated convection operators are to be evaluated within a cell, if a multiphysics or multiscale

model is implemented on the spacetree, or if the number of operations per element changes throughout the simulation, the weight attribute differs from leaf to leaf. In [23], the weights also depend on the surrounding elements: the more regular the environment, the smaller the workload becomes. An intelligent weight definition incorporates such facts.

It is cumbersome that the speedup exhibits a step pattern for regular grids. An improved parallelisation has to yield a smoothed-out speedup curve close to the linear speedup that is (almost) independent of the actual number of computing nodes. First studies working with overloading, i.e. several processes are started on each (multicore) computing node, yield promising results. In such experiments, I work with a logical number of computing nodes exceeding the physical number of nodes. Modern programming environments such as MPI 2 provide a dynamic process model, i.e. one can adopt the logical cardinality to the application's needs. With a reasonable extended number of logical computing nodes, the load balancing can search for a big, efficient number of nodes. Afterwards, it shrinks back the set of nodes to this efficient number.

With massive overloading, a sophisticated node pool realisation is essential. In the situation sketched above, a node pool tracking the relationship of logical and physical nodes helps to speed up the computation: It is absurd to make one cluster node hold several logical program instances, while other nodes employ only a single instance. The node pool is responsible to deliver the workers such that the load on the physical nodes is balanced. It can for example track the number of non-idle instances per physical node.

A sophisticated node pool furthermore takes the physical topology into account. If a cluster consists for example of several blades with several processing units per blade, some design decisions for such a sophisticated node pool strategy might be as follows:

- As Peano's logical topology equals a tree, it is reasonable to allocate workers for a node on the same blade: If a computing node tries to book a worker, the node pool searches whether a processor on the same blade is idle, and delivers this process, as this process can communicate with its new master without accessing the inter-blade connection typically being slower.

- On the other hand, it might, for the same reason, be reasonable to balance the working processes among the individual blades. Processors on one blade typically share network devices. If as few as possible processes on each blade are active, as few as possible processes have to share one piece of communication hardware.

These two examples highlight the interplay of the topologies. More sophisticated node pool strategies also take the partition boundaries and connections into account, as data are exchanged along the partition boundaries.

The parallelisation concept orbits around the load balancing problem. In the best case, it moreover considers memory considerations and topology issues. Long-running, massive parallel simulations nowadays have to face a much broader set of challenges: Reliability for example is a topic becoming more and more important since computational resources are limited and expensive, i.e. hardware failures, power blackouts, and breaks for high-priority jobs may not harm a simulation's result. Backup strategies and checkpointing algorithms face this challenge but are beyond the scope of this work. Yet, the interplay of stacks and space-filling curves yields an efficient serialisation of the partitions, and I am sure that checkpointing can benefit from them. Besides reliability, this work has not spent a single thought on efficient inter-program communication through standardised interfaces. The whole thematic block of simulation interaction paradigms for massive parallel codes offers a vast field of questions to be answered. Here, the inherent multi-scale character of this thesis is a promising link.

6 Numerical Experiments

In three (almost) othogonal chapters, this thesis addresses different challenges of high performance computing. Each chapter proposes algorithms, and each chapter also comprises an experiments section studying the effects and properties of these algorithms. Although the chapters are presented independent of each other, the idea of k-spacetrees underscores all of them. Because of this "leitmotiv", an application plugging the different thesis parts together should inherit all the nice individual properties directly: a fact to be validated. The following text studies such an application: a parallel, multigrid solver on adaptive Cartesian grids for the Poisson equation based upon Peano spacetrees, i.e. $(k = 3)$-spacetrees. All the experiments were conducted on the HLRB II.

6.1 Memory Requirements

A unique selling point of all spacetree codes is the low memory requirements to store a discretisation. As Peano's multigrid realises a matrix-free solver, i.e. it does not add an additional data structure for the matrices, e.g., this discretisation memory also governs the overall memory demands. One expects Peano to induce with very low memory requirements.

Three classes of properties at this determine the record size of each grid constituent: grid management data, solver- and PDE-specific data, and parallelisation entries. The grid management data comprises refinement information, spatial information, and state flags. It is discussed in Chapter 3. The PDE-specific data comprises current solution, PDE parameters such as the equation's right-hand side, error estimator, and helper variables. It is discussed in Chapter 4. The parallel data finally comprises domain partition information as well as load balancing, fork and join data. It is discussed in Chapter 5. Besides the grid constituents, it is also interesting to study the memory requirements of the traversal automaton. As Peano follows a recursive formulation, this automaton's memory footprint codetermines, in combination with the maximum k-spacetree height, the size of the call stack.

Peano's implementation comes along with a small number of bytes (Table 6.1). In comparison, out-of-the-box solvers supporting dynamic refinement, too, frequently require several hundred bytes per record. Furthermore, Peano's memory footprint is architecture independent, as I switched off any system-specific memory alignment.

Table 6.1: Memory requirements of the multigrid solver in bytes.

	Sequential		Parallel	
	$d = 2$	$d = 3$	$d = 2$	$d = 3$
Traversal automaton	40	56	40	56
Vertex	56	56	65	81
Geometric element	32	36	48	56
Vertex on output stream	20	20	52	68
Geometric element on output stream	10	14	34	42

Such an alignment speeds up the data access on current computer architectures. In the implementation, I thus use memory alignment for the internal data structures and the temporary data containers. For the input and output stacks, I disable the feature.

Records in the parallel mode exhibit an increased size. The geometric element then comprises additional load balancing data such as weight and δ attribute. The vertex comprises additional adjacency information, i.e. it has to track whether it is adjacent to remote spacetrees. If the latter holds, a vertex also has to hold the ranks of the associated processors, and this information is encoded by two adjacency lists with cardinality 2^d.

The precompiler DaStGen transforms all of Peano's records into a memory-optimised representation [13, 14] before they are passed to the compiler. This optimised representation uses for example one byte to hold all the individual bits and bit sequences encoding the refinement structure and vertex states. Nevertheless, it does not yet exploit the knowledge that most of the integers assigned to a vertex or cell have a bounded range. Integer numbers with a bounded range can also be compressed by DaStGen, and such a tuning reduces the memory consumptions further.

Even without the additional compression, Peano comes along with very modest memory requirements in both sequential and parallel mode although it supports arbitrary adaptivity. The Poisson solver inherits the memory properties already studied in Section 3.6.

6.2 Horizontal Tree Cuts

In Section 3.5.2, I introduce the concept of horizontal tree cuts swapping a part of the k-spacetree to a background buffer while the traversal continues to traverse the remaining, coarser spacetree levels. The multigrid's V-cycle in Chapter 4 then establishes a concrete application of this tree cut mechanism: As soon as the cycle has updated a fine grid level and has restricted the fine grid contributions to a coarser

grid, the multigrid's state machine reduces the active level, and all the events on the old fine grid level reduce to no operation. It is thus reasonable to swap this fine grid level to a background buffer until the V-cycle descends again into the level of interest. One expects Peano's multigrid solver to profit from the horizontal tree cuts in terms of runtime.

The experiments studying the tree cuts compared the runtime of 20 $V(2,2)$-cycles for both a regular grid for problem (4.30) and a grid resolving a singularity. The latter resulted from the homogeneous Poisson equation on the unit square with a right-hand side given by the Dirac distribution in the coordinate system's origin. This grid was extremely refined around the singularity, i.e. it refined solely the vertex at $x = 0 \in \mathbb{R}^d$ on each level.

Table 6.2: Speedup due to horizontal tree cuts.

	Minimal mesh width	Depth	Vertices	Speedup
regular, $d = 2$	$1.0 \cdot 10^{-1}$	4	$1.29 \cdot 10^3$	1.00
	$1.0 \cdot 10^{-2}$	6	$7.02 \cdot 10^4$	3.27
	$1.0 \cdot 10^{-3}$	8	$5.41 \cdot 10^6$	4.62
adaptive, $d = 2$	$1.0 \cdot 10^{-4}$	10	$4.99 \cdot 10^2$	1.00
	$1.0 \cdot 10^{-8}$	18	$9.39 \cdot 10^2$	1.67
	$1.0 \cdot 10^{-12}$	27	$1.73 \cdot 10^3$	2.00
regular, $d = 3$	$1.0 \cdot 10^{-1}$	4	$3.60 \cdot 10^4$	1.87
	$2.0 \cdot 10^{-2}$	5	$1.59 \cdot 10^7$	4.52
adaptive, $d = 3$	$1.0 \cdot 10^{-3}$	8	$3.40 \cdot 10^3$	1.38
	$1.0 \cdot 10^{-4}$	10	$4.37 \cdot 10^3$	1.38
	$1.0 \cdot 10^{-8}$	18	$8.25 \cdot 10^3$	1.51
	$1.0 \cdot 10^{-12}$	27	$2.68 \cdot 10^4$	3.07

The results in Table 6.2 prove the tree cut mechanism to be robust, i.e. the cuts never introduce a runtime penalty, and the speedup is always greater than one. Furthermore, the finer the grid becomes and the greater the number of vertices, the better the speedup due to the tree cuts. Consequently, the runtime improvement per grid hierarchy also scales with the spatial dimension d.

A comparison of the adaptive grids with the regular refined meshes reveals that the speedup per additional spacetree level depends on the regularity of the underlying grid, i.e. rather regular grids benefit more significantly from the feature than adaptive grids. This insight is not surprising, as the number of additional spacetree nodes per additional level increases by a factor of k^d for the regular grid. For the adaptive mesh resolving a singularity, the number of additional geometric elements or vertices, respectively, per additional level is fixed (see also Figure 5.16). Having a more flexible, local tree cut mechanism as discussed in the outlook of Chapter 3

(see Figure 3.21), the improvement for adaptive grids might increase, too.

As the tree cuts are robust with respect to runtime, there is not reason to switch this feature off anytime. Furthermore, the experiments revealed exactly the insights expected for a V-cycle.

6.3 Simultaneous Coarse Grid Smoothing

In Section 4.5.1, I introduce the simultaneous coarse grid smoothing extending the standard Jacobi smoother to a Jacobi smoother for adaptive Cartesian grids. The rationale behind this extension is that the Peano traversal processes all k-spacetree elements anyway. The experiments in Chapter 4 prove that the number of iterations reduces due to the additional coarse grid smoothing if the grid is adaptive. They nevertheless lack runtime measurement. If the additional smoothing operations on the coarse grid slowed down the individual steps of the V-cycles significantly, the simultaneous coarse grid smoothing could slow down the overall computation although it reduces the number of cycles. Hence, the runtime behaviour has to be studied carefully.

The experiments solved problem (4.33) on the unit square with a refinement criterion based upon the full stencil (4.22). The refinement threshold was reduced step by step, and I compared an $F(2,2)$-cycle with the simultaneous coarse grid smoothing with a plain $F(2,2)$-cycle working exclusively on one grid level a time. As the coarse grid smoothing's effect on the convergence rate has already been studied, each solver performed a fixed number of 20 cycles, and I concentrated on the pure execution time measured by hardware counters. Since the simultaneous coarse grid smoothing alters the numerics, both approaches yielded slightly different refinement patterns. I neglected these differences here.

Table 6.3: Runtime of the simultaneous coarse grid smoothing. No/yes denotes whether the solver smooths grids coarser than the active level, too. h gives the spacetree's height.

	$d=2$				$d=3$			
Threshold	h	Vertices	No	Yes	h	Vertices	No	Yes
$1.0 \cdot 10^{-4}$	6	$6.07 \cdot 10^3$	2.83s	2.96s	7	$7.22 \cdot 10^5$	660.35s	695.03s
$1.0 \cdot 10^{-5}$	7	$4.44 \cdot 10^4$	18.67s	19.50s	8	$1.45 \cdot 10^7$	12772.72s	13791.57s
$1.0 \cdot 10^{-6}$	9	$5.32 \cdot 10^5$	261.51s	278.65s				
$1.0 \cdot 10^{-7}$	10	$5.74 \cdot 10^6$	2575.86s	2763.99s				
$1.0 \cdot 10^{-8}$	11	$5.74 \cdot 10^7$	22937.93s	24510.41s				

The additional operations on the coarser grids increase the runtime per cycle by approximately ten percent (Table 6.3). This performance penalty factor is indepen-

dent of the problem size. It is thus reasonable to use the coarse grid smoothing all the time: for a given accuracy to be obtained, the solver without coarse grid smoothing always needs more than 110% of the cycles of the solver with the coarse grid smoothing.

6.4 MFlops on Regular Grids

Section 3.6 states an almost constant runtime per vertex for the pure grid management. One expects Peano's multigrid solver to come along with constant cost per vertex ratio, too.

The percentage of the peak performance achieved is one important quality metric for the quality of the implementation of a scientific software, as it measures how good an implementation fits to a given hardware. Besides time per vertex, this section thus also expresses the cost per vertex in terms of MFlops. A big set of tuning and optimisation strategies improving the MFlop rate exists. To select the right one, it is important to know in advance which effect prohibits the processing unit to display its full power. Cache effects are a popular first guess for such a candidate. Section 3.4 postulates that Peano's cache behaviour though is by construction asymptotically optimal, and the experiments afterwards prove this statement for the pure grid management. However, such a statement has to be verified for a real solver, too: One expects Peano's multigrid solver to be cache-oblivious.

One of Peano's unique selling points is the support for arbitrary dynamic, adaptive grids without memory drawbacks. However, it is also important to quantify how such a flexible solver performs for a regular mesh in terms of runtime and memory. The memory issue is already tackled by Section 6.1, and the next experiments thus concentrate on measurements for regularly refined spacetrees. They compare the runtime of $V(2,2)$-cycles to $F(2,2)$-cycles for the experiment (4.32) on the L-shape.

In the experiments, I sampled hardware counters for $V(2,2)$-cycles and $F(2,2)$-cycles. Besides the pure MFlop rates, the counters also tracked the number of L2 and L3 cache misses. As the counters' values were averaged over the complete runtime, the figures can not distinguish between grid setup and computation phase for the $V(2,2)$-cycle. Nevertheless, with 20 iterations conducted per experiments, the grid setup phase was assumed to be insignificant.

With the Itanium system and its $31 \cdot 10^{-3}$ Flop per cycle yielding approximately 50 MFlops, the measurements reveal the following insights: In the two-dimensional experiment, the MFlop rate raises to something around 70 MFlops with increasing problem size. Level two cache hit misses almost disappear throughout the measurements, while the level three cache hit rate drops with a finer resolution. The time spent on a vertex is almost constant for the V-cycle, and it exhibits a decline towards a lower bound of something around $2.5 \cdot 10^{-4}$ for the F-cycle. In the three-

Table 6.4: Hardware counter measurements for regular refined k-spacetree grids on L-shape.

	Vertices	$\frac{Flop}{Cycle}$	L2 hit rate	L3 hit rate	$\frac{t}{\#vertices}$
$V(2,2), d=2$	$1.19 \cdot 10^3$	$27.04 \cdot 10^{-3}$	99.98%	68.12%	$\ll 10^{-4}$
	$7.21 \cdot 10^3$	$35.88 \cdot 10^{-3}$	99.96%	97.02%	$\ll 10^{-4}$
	$5.48 \cdot 10^4$	$41.24 \cdot 10^{-3}$	99.95%	96.71%	$3.87 \cdot 10^{-4}$
	$4.63 \cdot 10^5$	$42.79 \cdot 10^{-3}$	99.95%	55.58%	$3.77 \cdot 10^{-4}$
	$4.08 \cdot 10^6$	$43.32 \cdot 10^{-3}$	99.95%	59.16%	$3.78 \cdot 10^{-4}$
$F(2,2), d=2$	$1.19 \cdot 10^3$	$27.15 \cdot 10^{-3}$	99.98%	60.38%	$\ll 10^{-4}$
	$7.21 \cdot 10^3$	$36.25 \cdot 10^{-3}$	99.96%	94.72%	$\ll 10^{-4}$
	$5.48 \cdot 10^4$	$41.79 \cdot 10^{-3}$	99.95%	96.41%	$2.98 \cdot 10^{-4}$
	$4.63 \cdot 10^5$	$43.48 \cdot 10^{-3}$	99.95%	55.88%	$2.81 \cdot 10^{-4}$
	$4.08 \cdot 10^6$	$42.72 \cdot 10^{-3}$	99.95%	40.53%	$2.71 \cdot 10^{-4}$
	$3.66 \cdot 10^7$	$42.97 \cdot 10^{-3}$	99.95%	36.96%	$2.52 \cdot 10^{-4}$
$V(2,2), d=3$	$3.50 \cdot 10^4$	$66.28 \cdot 10^{-3}$	99.96%	97.76%	$5.60 \cdot 10^{-4}$
	$6.20 \cdot 10^5$	$88.52 \cdot 10^{-3}$	99.96%	59.34%	$5.90 \cdot 10^{-4}$
	$1.42 \cdot 10^7$	$97.55 \cdot 10^{-3}$	99.97%	51.98%	$4.03 \cdot 10^{-4}$
$F(2,2), d=3$	$3.50 \cdot 10^4$	$66.61 \cdot 10^{-3}$	99.96%	97.03%	$5.08 \cdot 10^{-4}$
	$6.20 \cdot 10^5$	$89.20 \cdot 10^{-3}$	99.96%	59.18%	$5.01 \cdot 10^{-4}$
	$1.42 \cdot 10^7$	$97.73 \cdot 10^{-3}$	99.97%	51.50%	$4.92 \cdot 10^{-4}$

dimensional experiment, the MFlop rate raises to something around 160 MFlops, i.e. the MFlop rate of the two-dimensional experiment is more than doubled. The L2 and L3 cache behaviour of the two-dimensional experiments is preserved, while the cost per vertex is—besides the last V-cycle experiment—almost constant.

Peano's multigrid solver preserves a constant time per vertex model for both dimensions, and the multigrid solver preserves the grid traversal's cache characteristics. Hence, the statements from Chapter 3 also hold for the Poisson solver, and the multigrid code is cache-oblivious. Latter observation implies that techniques such as the file stack swapping can be directly applied to the Poisson solver, too.

The framework achieves less than 2.5 percent of the theoretical peak performance of one core of the HLRB II, while the switch from two to three dimensions just doubles the runtime per vertex. Studying solely the number of floating point operations—they are given by the local matrix sizes assigned to the elements' events—in terms of d, this increase appears to be small. Both properties deserve an explanation.

First, I spent no effort on a exhaustive source code optimisation of the Peano framework. This fact takes its toll.

Second, the partial differential equation and finite element method employed here are unsuited to achieve good MFlop rates. The grid traversal processes one element a time. Due to the checks from which stack to take the vertices from, due to the

checks whether the element and the vertices have to perform a transition, and due to the checks which events have to be triggered for which element constituents, each element processing comes along with a big number of case distinctions and integer arithmetics. The number of floating point operations per event though is small. A more complicated (system of) differential equations or a higher order method increase the computational load per element and, thus, lead automatically to a better megaflop rate due to a better usage of the superscalar computing facilities of the processor. For the Navier-Stokes equations, e.g., exactly the same code delivers four up to five times better rates [59], and the linear increase in computing time per vertex throughout the switch from $d = 2$ to $d = 3$ confirms this statement. It would not hold, if the caches or the memory system were the bottleneck.

Finally, the results show how important the elimination of all the integer arithmetics and case distinctions is. While the arbitrary adaptivity and hierarchical data structure is convenient for sophisticated mathematical algorithms, current computer architectures perform best, if the floating point data is sequentially filled without intermission into the floating point units. For regular grids studied here, such a streaming is theoretically possible—there is no need for refinement checks or the support of adaptive refinement—although it does not fit to Peano's traversal paradigm. First studies with a modified Peano implementation treating whole regular subgrids (patches) as one big stream, loading them en bloc without case distinctions, and passing them back to the output stream en bloc consequently exhibit an improved MFlop rate [23].

6.5 MFlops on Adaptive Grids

The previous section extensively discusses the interplay of cache hit rates, MFlop rates, and upcoming optimisations for regularly refined grids. This section studies the same solver properties for adaptive Cartesian grids. Section 3.6.1 suggests that the runtime cost per vertex remains the same besides a small, fixed overhead. To validate this assumption, the computational load per hanging vertex and the computational load to evaluate the refinement criterion have to be bounded, too: One expects Peano to exhibit constant runtime cost per vertex for adaptive grids.

Again, the experiments measured the hardware counters for experiment (4.33) solved by an $F(2, 2)$-cycle. In accordance with the results of Section 6.3, I switched on the simultaneous coarse grid smoothing throughout all the experiments. All the setups started from a coarse grid with four degrees of freedom, i.e. four inner vertices. The refinement criterion (4.22) then refined the tree until the linear surplus fell below a given threshold.

With such a setup, the following measurements and insights (Table 6.5) arise: In the two-dimensional case, the Flop per cycle ratio of the experiments with the

Table 6.5: Hardware counters for $F(2,2)$-cycle on adaptive grid with dynamic refinement criterion.

	Threshold	h	Vertices	$\frac{Flop}{Cycle}$	L2 hit rate	L3 hit rate	$\frac{t}{\#verices}$
$d=2$	$1.0 \cdot 10^{-4}$	6	$6.07 \cdot 10^{3}$	$36.85 \cdot 10^{-3}$	99.96%	95.35%	$4.94 \cdot 10^{-4}$
	$1.0 \cdot 10^{-5}$	7	$4.43 \cdot 10^{4}$	$42.09 \cdot 10^{-3}$	99.95%	97.84%	$4.41 \cdot 10^{-4}$
	$1.0 \cdot 10^{-6}$	9	$5.32 \cdot 10^{5}$	$42.95 \cdot 10^{-3}$	99.95%	53.29%	$5.23 \cdot 10^{-4}$
	$1.0 \cdot 10^{-7}$	10	$5.74 \cdot 10^{6}$	$42.21 \cdot 10^{-3}$	99.95%	38.94%	$4.81 \cdot 10^{-4}$
	$1.0 \cdot 10^{-8}$	11	$5.75 \cdot 10^{7}$	$42.56 \cdot 10^{-3}$	99.95%	37.24%	$4.26 \cdot 10^{-4}$
$d=3$	$1.0 \cdot 10^{-4}$	7	$7.13 \cdot 10^{5}$	$95.03 \cdot 10^{-3}$	99.97%	57.31%	$9.75 \cdot 10^{-4}$
	$1.0 \cdot 10^{-5}$	8	$1.45 \cdot 10^{7}$	$104.66 \cdot 10^{-3}$	99.97%	49.25%	$9.52 \cdot 10^{-4}$

regular grid is preserved. In the three-dimensional case, this rate even jumps over $104.66 \cdot 10^{-3} \frac{Flop}{Cycle}$. The cache hit rates of both test setups preserves the, meanwhile familiar, pattern already discussed in Section 3.6. Albeit the Flop rates do not change, the cost per vertex approximately double.

As the Flop rate remains roughly the same, the creation of the hanging vertices, the additional interpolations arising for them, and the refinement criterion evaluation do not stall the floating point pipeline. Nevertheless, they impose an additional overhead in terms of floating point operations. Consequently, the Flop rate is preserved, but the time per vertex increases.

As this increase is bounded by a factor of two, and as this experiment already exhibits real-world character—it is not an artificial grid or a grid resolving solely a singularity—Peano's cost per vertex are invariant for adaptive grids, too.

6.6 Outlook

The preceding sections study Peano's characteristics in detail. Throughout these studies, two properties become apparent: Due to the framework approach, characteristics demonstrated for the grid management apply directly to the matrix-free solver implemented on top of Peano. Due to the orthogonality of the individual chapters, the individual characteristics do not conflict with each other. Measuring the cache hit rate for the grid management, e.g., proved the framework to be cache-oblivious. The cache hit rate of the Poisson solver mirrors this characteristics. Furthermore, the Poisson solver's dynamic adaptivity criteria did not alter the cache behaviour: it is orthogonal to this property.

Besides the memory table on page 202, the experiments before though ignore the parallel version of Peano and concentrate solely on the sequential code. The orthogonality of the characteristics and the framework paradigm justify such a procedure, and it can be assumed that the parallelisation plugs seamlessly into the Poisson

solver. Nevertheless, a validation of this arguing is desireable. Furthermore, several properties of the parallelisation appraoch such as the dynamic load balancing show to advantage only within the context of a reasonable PDE. For a lack of time, I have to postpone such studies to further work.

7 Conclusion

In this thesis, I introduce the framework Peano for grid-based solvers of partial differential equations. Any such endeavour has to face one particular question: is it worth launching yet another framework? In my opinion, Peano derives its right to exist from a combination of advantageous characteristics shaping any solver programmed on top of the framework:

First, the solvers' grid management requires a low amount of memory. Few bits per vertex or geometric element, respectively, are sufficient due to the depth-first traversal with a corresponding tree encoding. While solvers on Cartesian grids and solvers using patches of Cartesian grids have a modest memory demand, too, they usually do not allow for an arbitrary, dynamic refinement of the grid.

Second, the framework supports meshes exceeding the available main memory without a runtime penalty due to a stack-based persistence concept where the main memory acts as cache for the hard disk. While alternative hard disk swapping strategies might provide a similar runtime behaviour, they usually pose restrictions on the grid layout whereas the swapping here is completely encapsulated.

Third, the solvers' grid management supports a multiscale geometric representation of the domain due to the k-spacetrees. While alternative multiscale discretisations provide such data by definition as well, few allow the user to refine and coarse the grid arbitrarily without changing the data topology underlying the grid storage.

Fourth, the solvers exhibit constant runtime cost per record and these are independent of the actual grid resolution and layout due to a cache-aware or memory-hierarchy-aware, respectively, grid traversal benefiting from the interplay of stacks with space-filling curves. While many cache optimisations yield a similar cache hit rate, their memory behaviour usually breaks down if the grid changes dynamically or becomes adaptively refined.

Fifth, the solver's grid management supports a domain decomposition providing domains with small surfaces compared to their volume due to space-filling curves. While many decomposition approaches exhibit nicely shaped subdomains and realise the accompanying data exchange efficiently, these shapes typically suffer from dynamically changing discretisations. Furthermore, few domain decompositions take into account the grid hierarchy explicitly. Peano does.

Before I continue, one remark: Many papers promote space-filling curves enthusiastically. One often can not help but think that these curves render all the parallelisation challenges null and void, as they make the search for a good partitioning

simple—just cut the whole curve into equally sized pieces—and yield quasi-optimal partitions on-the-fly. I do not agree. Naive equally-sized cuts of space-filling curves do not take multiscale relations into account; adaptive grids and multiscale algorithms with inter-level communication are not tracked, but need a global representative of the whole domain; moving or relocating cuts along the curve is, on the one hand, far from trivial due to the complex subdomain shapes and structures, as well as, on the other hand, not for free because of the data to be exchanged. Parallelism remains hard work.

As final characteristics, Peano's parallelisation incorporates a dynamic load balancing due to a simple tree attribute grammar. While many load balancing algorithms produce equivalent or even better workload distributions, few work on-the-fly, in parallel, consider a multiscale representation of the domain, and are near as simple.

As far as I know, no other framework provides all these features in one package. The key to obtain them is the mixture of spacetrees, space-filling curves, and a refinement-invariant, stack-based—the number of stacks is linear in d and independent of grid structure and refinement depth—grid storage scheme.

Besides functional properties, the handling of complexity, maintainability, and extendability accompanies any discussions on software frameworks and software architectures. One technique for tackling these three aspects is encapsulation together with separation-of-concerns. Frameworks hiding technical details and their realisation from the code solving the actual problem fit into this dogma. In high performance computing, the quality of a software however is determined by two different non-functional requirements: execution speed and parallel scalability. As functional decompositions often introduce runtime bottlenecks, programmers of simulation codes often downgrade the separation of concerns aspect and interweave grid- and cluster-specific as well as PDE-specific code fragments. Peano tackles the non-functional challenges with a rigorous object-oriented architecture and a simple, event-based coupling of PDE-specific parts with the grid and traversal management. This event paradigm, on the one hand, enforces a separation of PDE-related tasks from the well-structured framework. On the other hand, they enable the framework to schedule activities in-between any two PDE tasks, i.e. the paradigm facilitates a mixture of different activities.

Whether a framework fulfills a functional or non-functional requirement or whether it is extendable to fulfill a requirement has to be studied by means of concrete applications built atop the framework. Furthermore, the collection of all these applications sheds light on the debate how valuable a framework is for answering and addressing unsolved questions from science and engineering. In the following, I hence first present a solver for computational fluid dynamics (CFD) that is implemented with Peano. While this solver exhibits the highest maturity of any software implemented with the framework, it does not yet exploit all the functional properties

coming along with Peano. I thus continue with a list of interesting properties and features upcoming PDE solvers could or will provide—some efforts already started, some not. From this discussion, I return to the non-functional aspects. Running software nearer and nearer at the (multicore) peak performance is of the essence of high performance computing. I highlight one crucial extension and evolution ansatz that has already shown to be promising to tackle the runtime challenge. Having functional and non-functional look-outs and extensions at hand, I finally pick up again a set of philosophical questions: What programming and software strategy is well-suited for future-generation multiphysics, multiscale, massive parallel high performance codes? Is a holistic top-down approach represented by frameworks or is a component-based decomposition of the challenges the way to go? Elaborating a recently prototyped integration concept for Peano, I combine advantages from both worlds and embed Peano into the whole simulation pipeline.

Computational Fluid Dynamics with Peano

A framework for PDE solvers is as good as the best PDE solver implemented with the framework. Fortunately, Peano has a sophisticated computational fluid dynamics code [60] built on top of it. This solver is actually a collection of plugins addressing the stationary and instationary, incompressible Navier-Stokes equations via direct numerical simulation. The solver suite provides an exhaustive set of standard boundary conditions, and it also supports fluid-structure interaction scenarios due to a separated approach, i.e. the CFD code can compute forces introduced by the fluid at the domain's boundaries, and it can handle moving boundaries in return. Multiple spatial discretisations of the equations facilitate a spatial discretisation of higher order—which is an interesting fact for any PDE solver, as it follows the appeal to increase the "science per FLOP" [44] or "science per record" as well as Flop per event (Section 6.5)—and the suite provides several implicit and explicit time integration schemes.

The overall goal of the CFD plugins is to show that interesting insights and opportunities stem from the crossfire of adaptive Cartesian grids and computational fluid dynamics. In this short outline, I concentrate on the CFD solver's properties that are interwoven with the framework, i.e. I neglect characteristics that are not directly related to Peano.

First, the simulation can resolve the computational domain very accurately due to the low memory requirements of the Peano framework. Such a fine resolution is for example important if a simulation computes turbulent regimes with direct numerical simulation, i.e. without explicit turbulence models. Here, the turbulent behaviour on the scale of interest stems from fine scale influences. Out-of-the-box CFD codes often can not afford to resolve these fine scales and, hence, have to apply turbulence

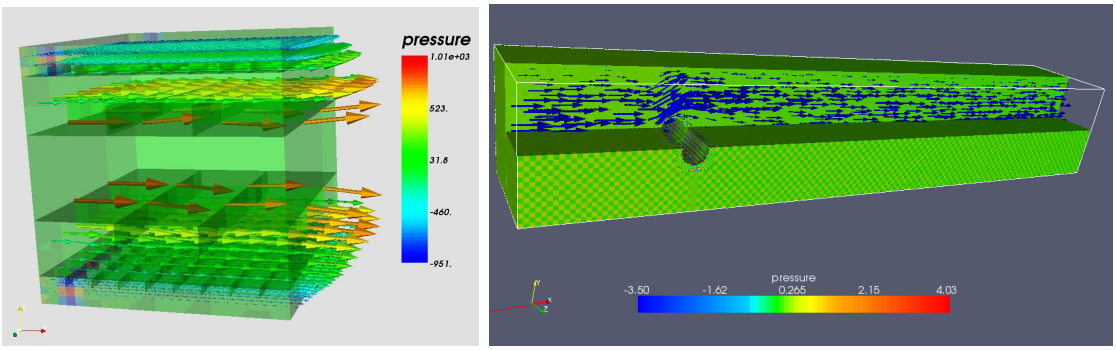


Figure 7.1: Channel filled with gas (left) and fluid circulating around cylinder (right). Illustrations are due to Tobias Neckel [60] who realised a computational fluid dynamics code within the Peano framework.

models to imitate the fine grid influences. Peano's direct numerical simulation can validate these models and vice versa.

Second, the adaptive grids can resolve selected regions significantly finer than the overall domain due to the low memory requirements and the support of arbitrary adaptivity. Such an improved resolution is for example important if a simulation computes flows with different characteristics such as boundary layers where the overall flow is turbulent whereas the flow along the boundaries is laminar (Figure 7.1). Here, interesting effects arise from the interplay of the two different types of flow, and the need for fine grid resolutions is motivated by physics. Out-of-the-box CFD codes often can not afford to resolve the interesting regions with a sufficient accuracy.

Third, the dynamic adaptivity can adapt the grid to a (changing) geometry due to the support of dynamic refinement. Such a dynamic change of the computational grid is for example important in a fluid-structure interaction problem, where the structure, i.e. the computational domain, permanently changes its position or shape and is not aligned with the coordinate system axes. Out-of-the-box CFD codes often can not adopt the grid arbitrarily and have to stop the simulation after a given number of time steps, have to remesh, and, finally, interpolate from the old grid to the new grid. Peano provides a homogeneous environment to handle such flows and computational domains where the mesh update accompanies each single time step.

Besides the three applications highlighted here, the implementation of [60] emphasises several highlights not correlated directly with the framework. Besides them, it also adds for example a restart functionality to Peano's pool of features. Restarts due to regular snapshots are important for long running parallel computations, where the malfunction of one single node would destroy the overall computation's result.

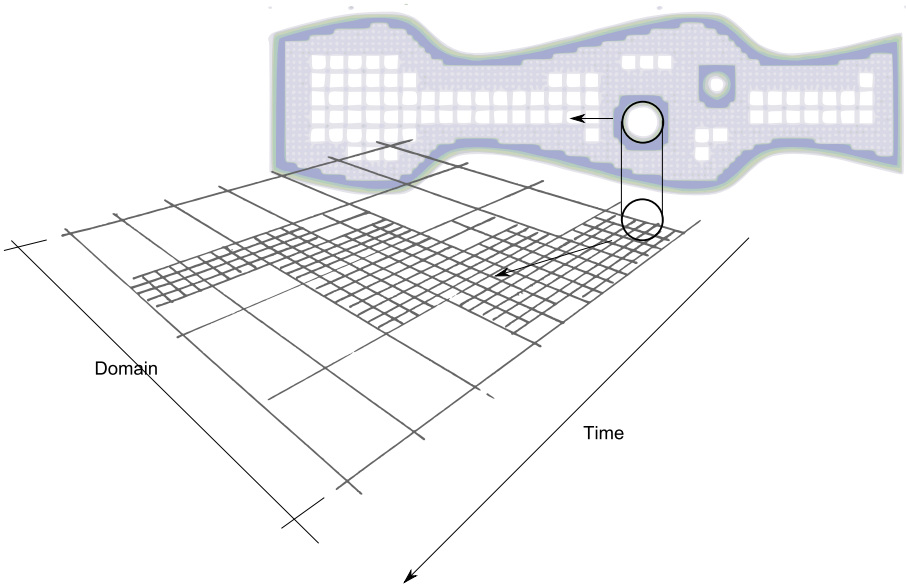


Figure 7.2: Space-time grid with adaptive refinement in both time and space. Illustration of simulation snapshot is due to Bernhard Gatzhammer who uses Peano's CFD realisation within a fluid-structure interaction application environment.

Furthermore, the snapshot mechanism enables the user to run a simulation for a given time. Afterwards, he can re-setup the application's parameters and conduct several parameter studies on the same startup setting.

New Applications Using the Peano Framework

Computational fluid dynamics is an active and an agile field of research. Many fundamental questions in this field are still open, and the computational challenges are far from being solved. Peano's CFD implementation is a promising candidate to become part of the active research. Nevertheless, the CFD implementation does not yet exploit all of Peano's unique selling points. In the text underneath, I hence pick up two additional application areas and concentrate on additional properties.

Peano allows a mathematical model to use an arbitrary $d \geq 2$. While the CFD code selects $d \in \{2, 3\}$, searching for models with a bigger dimension constant yields a vast number of possible applications: Parameter studies and parameter fitting and optimisation problems for example add additional parameter dimensions to classical fluid dynamics problems. Some problems from mathematical finances are by construction based on a higher dimensional computational domain. I concentrate

on a different class of "high-dimensional" partial differential equations.

Many parabolic equations are given on a computational domain with dimension three and an additional time dimension. Choosing $d = 4$, Peano can solve such an equation treating the time as just another parameter dimension of the computational domain (Figure 7.2). In first experiments with simple setups for the heat equation, such a holistic approach with space-time discretisations brings along at least three interesting opportunities: First, problems with periodic solutions benefit from the fourth simulation dimension. Starting with a first guess on a (rather coarse) time grid, the simulation successively improves the solution and adopts the grid, the time stepping, and the periodic boundary conditions in time. No additional effort is to be spent on the treatment and persistence management of the individual time steps, and the parallelisation falls seamlessly into place. Second, problems with multiscale temporal behaviour benefit from the fourth simulation dimension. Many solutions exhibit regions where the solution changes rapidly, while other parts of the computational domain remain almost invariant. Here, the solution is very smooth in time. A locking of the time steps leading to a global time stepping treats both types of regions the same. Adaptive time stepping approaches, where regions of interest are tracked with smaller time steps, fit to the four-dimensional discretisation, as the $(d = 4)$-spacetree simplifies the persistence management of the individual time steps and the interpolation in time. Finally, simulations with strong pollution effects in time benefit from the fourth simulation dimension. For parabolic simulations, errors in one time step propagate to the subsequent time steps, spread spatially, and can break down the overall simulation accuracy. A prominent candidate for such a misbehaviour is the preservation of mass in a fluid-structure simulation with incompressible fluids: instationary boundaries introduce small mass inconsistencies around moving geometries and these inconsistencies then spread globally. The four-dimensional grid simultaneously holds all time steps. Whenever pollution is identified, it is thus possible to return to the time step causing the pollution, refine the grid there locally, and update all subsequent time steps. The pollution's source is eliminated.

Besides the higher-dimensional formulations, the multiscale representation of the domain deserves an additional remark. The Poisson equation studied in this thesis and the CFD code introduced above rely on a holistic description of all the physics by a single partial differential equation, i.e. the solution is determined by one equation valid for all spatial resolutions. A promising field for new scientific insights is the coupling of different mathematical models for different spatial scales. It is for example a common wisdom that the boundary conditions coming along with classical fluid dynamics codes are often an inappropriate description of the real world. In such a case, boundary conditions arising from molecular dynamics yield more accurate global simulation results (Figure 7.3). Yet, noone can afford to compute large-scale flows with a molecular dynamics code. Peano's multiscale discretisation

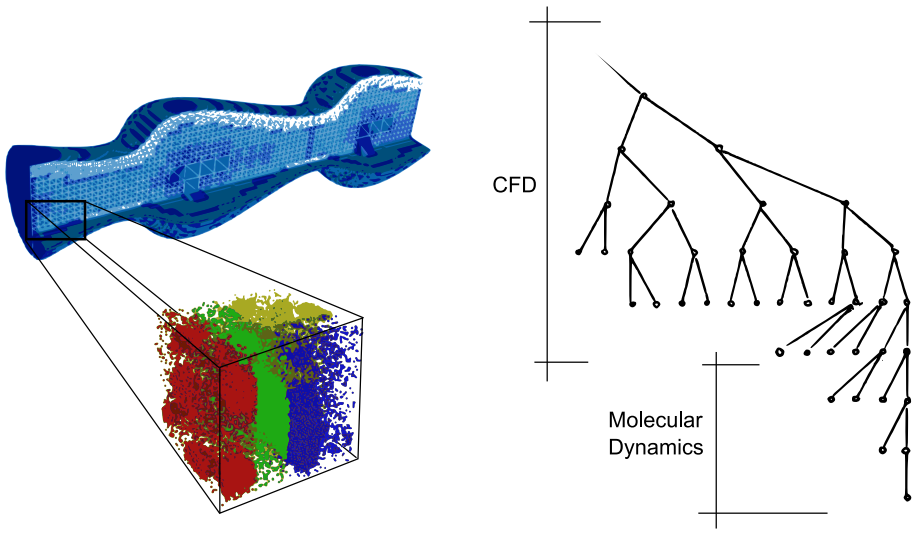


Figure 7.3: Boundary condition of sophisticated fluid simulation (top) results from molecular dynamics code (bottom). Illustration of molecular dynamics is due to Martin Buchholz.

makes it possible to embed different physical models and mathematical descriptions into one code, couple them strongly in regions of interest—only regions of interest are refined such that the mulecular dynamics code comes into play—and make both simulations benefit from each other.

Framework Extensions

This chapter so far concentrates on use cases and extensions concerning different functional properties of Peano from an application point of view. Besides a functional evolution of Peano's plugins, emphasis has also to be put on non-functional properties, as the applicability and quality of a framework depends to a great extend on the maintainability, extendability, and performance of the code. The following paragraphs refrain from a concrete application and concentrate on technical extensions and improvements reducing the software's runtime.

Performance considerations particularly gain weight with the new hardware architectures arising: Future architectures consist of more and more cores, while the single-core performance remains almost constant. Code then does not anymore automatically benefit from new architectures due to increased clock rates [71]. In accordance, performance metrics also undergo a fundamental paradigm shift. They incorporate both percentage of peak performance achieved and scalability with respect to multicores.

The tuning of Peano is beyond the scope of this thesis, and, besides the cache discussion, no significant emphasis is put on the debate whether Peano fits to and how it performs on current computer architectures. As a result, Peano is a framework with constant cost per degree of freedom and a vast amount of features. Yet, it performs rather poor with respect to the theoretical peak performance of a processing unit. This has to change with upcoming framework releases.

Methodologically, I consider recursion unrolling [65] as fundamental technique to make Peano utilise current computer architectures. In first case studies [23], an additional attribute grammar similar to the weight and δ attribute tracks which subtrees of the k-spacetree are invariant and correspond to regular refined subgrids. On these subgrids or patches, respectively, the code switches from the recursive traversal to a flat, sequential implementation holding the complete subtree as one sequence of plain Cartesian multiresolution grids.

Having whole Cartesian grids at hand allows for Peano to apply at least three fundamental performance improvements. First, the regular, homogeneous data structure with a sequential processing fits to instruction level parallelisation as soon as operations on these records exploit SSE and vector processing units. Second, the regular, homogeneous data structure fits to a simple data decomposition cutting it into equally sized continuous pieces. Multicore architectures with multiple cores holding one shared block of data can then take over the responsibility of subparts of the patches. They share the computational workload. Finally, the regular, homogeneous data structure fits to block-wise numerical algorithms. Block-wise red-black Gauß-Seidel algorithms for example are easy to realise on these blocks, but improve the numerical performance, i.e. the convergence rate.

First experiments applying recursion unrolling yield promising results for grids with big regular subpatches, i.e. the convergence rate improves, the code scales on several cores, and the MFlops raise. Whereever a (sub)grid changes, exhibits adaptive discretisations, or covers a domain partition's boundary, the application falls back to the standard Peano implementation. With such a hypbrid implementation, Peano's MFlop rates improve by an order of magnitude, while the code preserves all the flexibility and features introduced in this thesis.

Software Challenges

Deriving new algorithmic extensions and innovations is fun, and every enthusiastic programmer appreciates the development of a system or a new algorithmic feature from scratch. A mere discussion of additional features arising from applications or performance optimisations nevertheless misses one crucial point of software development in high performance computing: Is the software's quality sufficiently high, and does the software remain maintainable, manageable for humans, and extendable?

At the time this thesis is handed in, the Peano framework in combination with the computational fluid dynamics plugin already comprises more than 850 classes distributed among 35 packages. It is hence reasonable, even indispensable, to ask such questions.

First of all, using a framework does not weaken this challenge. Good component decompositions provide a set of tools, represented by the components, with well-defined interfaces. The user then selects suitable components and combines them to an application. A good component decomposition neither forces the user into a dedicated programming and development style nor do the components influence each other directly—component decomposition does not coin the software development process, and it exhibits a low acceptance threshold. Frameworks, by contrast, require the user to make his or her programming approach and train of thought fit to the framework's paradigm, design, and philosophy. In turn, their holistic approach facilitates runtime and memory efficient implementations. A simple example: No matter how sophisticated and fast a solver component for linear equation systems and a component for the grid management are realised; if a discretisation changes permanently, and if the assembly of the linear equation system that analyses the grid is a complicated and long-running activity, the application can not benefit from the elaborate components it relies on. A clever integration and interweaving of components circumnavigates such a problem a priori—updating for example only matrix entries related to changing grid elements—and frameworks are one way to enforce this. For an easy adoption and a low acceptance threshold, it is though essential to design a framework's restrictions and paradigms as clear, plain, and straightforward as possible: It is unacceptable to force the programmer to read through and understand several Ph.D. theses, each with 222 pages or more, before programming starts.

Peano introduces the event concept to tackle this challenge for PDE solvers built atop the framework. While the events hide the complete persistence management and traversal realisation from the PDE programmer, the framework realisation itself follows a rigorous object-oriented design incorporating many best practices and design patterns. Nevertheless, features such as parallelisation or recursion unrolling are that profound—a complete decoupling from the framework and its signatures here is not possible anymore—that it has carefully to be studied whether a particular realisation complicates the events' signature or semantics (the level-wise depth-first traversal for example falls into this class) and whether it increases the framework's code complexity. Due to the clear functional decomposition, I consider the implementation at hand now to be maintainable, manageable, and extendable. However, preserving these properties becomes more complicated with each feature added.

In this context, the question arises whether an object-oriented language such as C++ is the right choice for the implementation. Object-oriented languages have been considered the magic bullet for a long time since their construction enforces a

rigorous encapsulation of details. For Peano, two drawbacks though appear: On the one hand, breaking down a design into fragments with one particular responsibility per type is not always possible: A grid vertex, e.g., combines PDE-specific properties, the management of the grid's adaptivity, and operations to exchange data in a shared memory environment due to message passing. For such a type, an aspect or feature oriented paradigm—the vertex is added technical functionalities such as message exchange operations automatically, and its signature and appearance depends on the component handling the record—are better suited. Yet, there is a lack of such tools for high performance computing. Peano exploits the precompiler DaStGen [13, 14] addressing the aspect oriented challenge. While such a tool is a first step, the complete tool chain, the feature oriented nature, and the underlying programming paradigm is neither fully understood nor perfected. On the other hand, Peano's code blocks often are difficult to understand because of the sophisticated mathematical formulas and algorithms they are realising. They consist of long sequences of loops, matrix-vector products, and complicated mathematical expressions. While C++'s generics due to expression templates attenuate this problem, domain-specific languages such as computer algebra systems or Matlab lead to the most compact, maintainable, and intuitive formulations of such code blocks.

With a hand-written, tailored precompiler for aspect-oriented programming, a rigorous encapsulation with object-oriented language constructs, and powerful expression templates, I consider the implementation at hand now to be maintainable, manageable, and extendable. Though, understanding of the underlying principles and overcoming (at least some of) these issues with something beyond C++ for high performance computing is highly desireable; in particular, when the software becomes more complex with each feature and application added.

Peano as a Component

The preceding two paragraphs are a plea for frameworks for PDE solvers, as they make the problems design and implementation challenges and as they explain them with inadequate language support. Such an argumentation in favour of frameworks is biased and falls short: A clear bottom-up component decomposition with well-defined responsiblities and states has proven of great value for many problems throughout centuries, and it eliminates many software challenges in advance as the resulting code is focused and typically smaller. For the grid-based solvers on dynamically adaptive meshes, I quote the performance as crucial advantage of an integrated approach where everything is done in one place, i.e. within the framework. As soon as the performance argument is not overwhelming anymore, there is no reason to dismiss a component architecture a priori in favour of a framework approach; particularly, since the threshold of acceptance for component architectures is typically

much lower due to smaller component size and complexity. Consequently, I suggest a twofold approach: Whenever performance is crucial, a feature is realised within Peano. Whenever performance is not crucial, a feature is deployed to an external component of its own. Peano as a result runs as one big component among several others. It incorporates all the performance critical features, and interacts with other components whenever I favour a clear separation of concerns over performance.

The geometry management is one example for a component: Peano's geometric events reduce to a set of "is inside" and "is outside" queries as well as a check whether and which boundary has been intersected by a hypercube. The latter information is important for a PDE solver to set the appropriate type of boundary conditions. While Peano could comprise a geometry management with mesh IO routines, surface management, and domain modificators (for fluid-structure interaction, e.g.), I prefer the geometry events to belong to one interface connected to an external component. At our chair, the tool preCICE for example implements such an interface. The resulting two-component architecture has several advantages: First, it follows a separation-of-concerns idea. Peano already incorporates a vast amount of ideas and technical details dealing with grid storage, grid traversal, parallelisation, PDE solvers, and so forth. There is no need to add additional complexity. Second, a component architecture allows the user to exchange the geometry realisations. For my experiments, few hard-coded geometries are sufficient, whereas the computational fluid dynamics plugin benefits from a geometry component that can handle complicated computational domains from external mesh files. Finally, a component interface provides a single-point-of-contact (SPOC) for Peano's parallel realisation: All the different Peano instances in a cluster contact one instance of the geometry component on one computational node. Consequently, the geometry data are always consistent and the dynamic parallelisation is hidden from the geometry component. First experiments with preCICE yield promising results with respect to separation-of-concerns, exchangability, and parallel environments.

While the component interfaces at the moment are plain C++ signatures coupled at compile-time by direct function calls or MPI messages, such a component interpretation of Peano fits perfectly to the common component architecture (CCA): To translate the C++ signatures to SIDL is straightforward, and the resulting Peano codes realising one or several PDE solvers can act as CCA component within a bigger problem solving environment. To have Peano's PDE solvers as components within a greater simulation workbench is an ultimative goal bridging the gap from codes that study individual properties prototypically to codes that are used for real-world problems on real-world data sets by application-domain experts.

In the End

This thesis provides an enormous number of links for future work: The individual chapters present improvements and extensions concerning the individual features, and this final chapter gives several more complicated and laborious improvements and extensions that embed Peano into a bigger environment of open questions, existing software packages, and computational challenges. Subsuming all these aspects with a few final sentences is impossible. I will pick up several issues throughout the upcoming years, as I believe Peano being a promising approach to obtain new, interesting, and powerful PDE solvers. Although it has several shortcomings—the peak performance perhaps being the most important one—introducing a lot of work to invest, there is also a lot of insight to harvest. In the end, the convenience of the effort will be measured by the understanding obtained with the tool—a statement holding for the whole discipline of computational sciences and engineering. The purpose is insight, not numbers.

A Helper Algorithms

The following pages provide some algorithms swapped from the original text to the appendix. They are moved to this chapter either due to their technical character or if they required for too much space and made the text fall into pieces. All the algorithms are used by the grid persistence management and traversal.

	Usage	Name
A.1	Page 70	$belongsToTouchedFace$
	Page 71	$belongsToUntouchedFace$
A.2	Page 76	$createSubStates$
	Page 84	
A.3	Page 76	$setExitManifold$
	Page 84	$setEntryManifold$
	Algorithm A.3	
A.4	Page 76	$removeFaceAccessNumber$
	Page 84	
	Algorithm A.5	
A.5	Page 76	$setFaceAccessNumber$
	Page 84	

Algorithm A.1 Determine for a vertex at *position* whether any adjacent face is touched, i.e. whether an element sharing a face with the current element has read this vertex before, or whether all adjacent faces are untouched, i.e. no element sharing a face with the current element has written the vertex before. The position corresponds to a lexicographic vertex enumeration within a hypercube.

$$\begin{aligned} belongsToTouchedFace &: \mathcal{P}^{2d}_{\text{touched}} \times \{0,1\}^d \mapsto \{\top, \bot\} \\ belongsToUntouchedFace &: \mathcal{P}^{2d}_{\text{touched}} \times \{0,1\}^d \mapsto \{\top, \bot\} \end{aligned}$$

```
1:  procedure belongsToTouchedFace(touched, position)
2:      result ← ⊥
3:      for i ∈ {0, ..., d − 1} do
4:          face ← i
5:          if position_i = 1 then
6:              face ← face + d
7:          end if
8:          result ← result ∨ touched_face
9:      end for
10:     return result
11: end procedure
12: procedure belongsToUntouchedFace(touched, position)
13:     result ← ⊥
14:     for i ∈ {0, ..., d − 1} do
15:         face ← i
16:         if position_i = 1 then
17:             face ← face + d
18:         end if
19:         result ← result ∨ ¬touched_face
20:     end for
21:     return result
22: end procedure
```

Algorithm A.2 The following algorithm derives both the *even* and the *access* flags for all geometric subelements of one refined element. It analyses the parent element's flags. The result tuples are enumerated along the leitmotiv. Algorithm relies on Algorithm A.3.

$$\begin{aligned} \mathcal{A} &:= \{-2d+1, \ldots, 2d-1\}^{2d} \\ \mathcal{E} &:= \{\top, \bot\}^{d} \\ createSubStates : \mathcal{A} \times \mathcal{E} &\mapsto (\mathcal{A} \times \mathcal{E})^{3^d} \end{aligned}$$

```
 1: procedure createSubStates(access, even)
 2:     return createSubStates(access, even, d − 1)
 3: end procedure
 4: procedure createSubStates(access, even, axis)
 5:     if axis = −1 then
 6:         return (access, even)
 7:     else
 8:         even0 ← even
 9:         even1 ← even
10:         even2 ← even
11:             ▷ Create three copies of even flags corresponding to three substates.
12:         even1_axis ← ¬even1_axis
13:                     ▷ Flag "in-between" has one entry differing from the others.
14:         access0 ← setExitManifold(access, even, axis)
15:         access1 ← setExitManifold(access, even, axis)
16:         access1 ← setEntryManifold(access1, even, axis)
17:         access2 ← setEntryManifold(access, even, axis)
18:                                 ▷ Set new entry/exit manifold for three substates.
19:         return
20:           ((createSubStates(access0, even0, axis − 1),
21:            (createSubStates(access1, even1, axis − 1),
22:            (createSubStates(access2, even2, axis − 1))
23:                                 ▷ Recursive calls and concatenation of results.
24:     end if
25: end procedure
```

Algorithm A.3 Two helper operations for Algorithm A.2 simplifying the *access* flag manipulation. They set a new entry or exit manifold and upate the *access* flag such that the invariants (3.5) and (3.6) hold again.

$$\begin{aligned} \mathcal{A} &:= \{-2d+1, \ldots, 2d-1\}^{2d} \\ \mathcal{E} &:= \{\top, \bot\}^{d} \\ setExitManifold : \mathcal{A} \times \mathcal{E} &\mapsto \mathcal{A} \\ setEntryManifold : \mathcal{A} \times \mathcal{E} &\mapsto \mathcal{A} \end{aligned}$$

```
1: procedure setExitManifold(access, even, axis)
2:     if isTraversePositiveAlongAxis(even, axis) then
3:         access ← removeFaceAccessNumber(access, axis + d)
4:                                                  ▷ see Algorithm A.4
5:         access ← setFaceAccessNumber(access, axis + d, 1)
6:                                                  ▷ see Algorithm A.5
7:     else
8:         access ← removeFaceAccessNumber(access, axis)
9:                                                  ▷ see Algorithm A.4
10:        access ← setFaceAccessNumber(access, axis, 1)
11:                                                 ▷ see Algorithm A.5
12:    end if
13:    return access
14: end procedure
15: procedure setEntryManifold(access, even, axis)
16:    if isTraversePositiveAlongAxis(even, axis) then
17:        access ← removeFaceAccessNumber(access, axis)
18:                                                 ▷ see Algorithm A.4
19:        access ← setFaceAccessNumber(access, axis, −1)
20:                                                 ▷ see Algorithm A.5
21:    else
22:        access ← removeFaceAccessNumber(access, axis + d)
23:                                                 ▷ see Algorithm A.4
24:        access ← setFaceAccessNumber(access, axis + d, −1)
25:                                                 ▷ see Algorithm A.5
26:    end if
27:    return access
28: end procedure
```

Algorithm A.4 Helper for Algorithm A.3. Invalidates *access* entry on $face$, i.e. *access*'s entries are shifted such that (3.5) and (3.6) hold again for all entries besides face's *access* entry. If face's entry $access_{face}$ is greater than zero, all other entries greater than $access_{face}$ hence have to be decremented. If face's entry $access_{face}$ is smaller than zero, all other entries smaller than $access_{face}$ hence have to be incremented.

$$\mathcal{A} := \{-2d+1, \ldots, 2d-1\}^{2d}$$

$$removeFaceAccessNumber : \mathcal{A} \times \{0, \ldots, 2d-1\} \mapsto \mathcal{A}$$

1: **procedure** $removeFaceAccessNumber(access, face)$
2: $oldAccessNumber \leftarrow access_{face}$
3: **if** $access_{face} > 0$ **then**
4: **for** $i \in \{0, \ldots, 2d-1\}$ **do**
5: **if** $access_i \geq oldAccessNumber$ **then**
6: $access_i \leftarrow access_i - 1$
7: **end if**
8: **end for**
9: **end if**
10: **if** $access_{face} < 0$ **then**
11: **for** $i \in \{0, \ldots, 2d-1\}$ **do**
12: **if** $access_i \leq oldAccessNumber$ **then**
13: $access_i \leftarrow access_i + 1$
14: **end if**
15: **end for**
16: **end if**
17: $access_{face} = 0$
18: **return** *access*
19: **end procedure**

Algorithm A.5 Helper operation for Algorithm A.3. Set a face's *access* entry to a new value. If this value is greather than zero, all other *access* entries greater than *value* are incremented. If this value is smaller than zero, all other *access* entries greater than *value* are decremented. As a result, the constraints (3.5) and (3.6) hold again.

$$\mathcal{A} := \{-2d+1, \ldots, 2d-1\}^{2d}$$

$$setFaceAccessNumber : \mathcal{A} \times \{0, \ldots, 2d-1\} \times \mathbb{N}_0 \mapsto \mathcal{A}$$

```
1: procedure setFaceAccessNumber(access, face, value)
2:     if value > 0 then
3:         for i ∈ {0, ..., d − 1} do
4:             if access_i ≥ value then
5:                 access_i ← access_i + 1
6:             end if
7:         end for
8:     else
9:         for i ∈ {0, ..., d − 1} do
10:            if access_i ≤ value then
11:                access_i ← access_i − 1
12:            end if
13:        end for
14:    end if
15:    access_face ← value
16:    return access
17: end procedure
```

B Hardware

The following sheets enlist the hardware used throughout the thesis. The experiments typically are conducted on up to four different architectures. All the measurements resulting from hardware counters were conducted on Itanium nodes. The chair's local Pentium machines are connected via a standard Ethernet, i.e. they do not belong to a high performance cluster.

Pentium	
Processor	Pentium 4
Vendor	Intel
Bit	32
Cluster	Local workstations
Location	Chair of Scientific Computing in Computer Science Technische Universität München
One core	
Peak performance	6.8 GFlop/s
Clock rate	3.40 GHz
Cores	1
Level 1 data cache	16 KByte
Level 2 cache	1 MByte
Level 3 cache	-
Memory	2 GByte
Cluster	
Nodes	-
Type	Ethernet

Opteron	
Processor	Opteron 850
Vendor	AMD
Bit	64
Cluster	Infinicluster
Location	Chair of Rechnertechnik und Rechnerorganisation Technische Universität München
One core	
Peak performance	4.8 GFlop/s per core
Clock rate	2.4 GHz
Cores	4 per node
Level 1 data cache	64 KByte, 64 Byte per line
Level 2 cache	1 MByte, 64 Byte per line
Level 3 cache	-
Memory	8 GByte shared between 4 cores
Cluster	
Nodes	32
Type	4X Infiniband

Itanium	
Processor	Itanium2 Montecito Dual Core
Vendor	Intel
Bit	64
Cluster	HLRB II—Höchstleisungsrechner Bayern II SGI Altix 4700
Location	Leibniz Supercomputing Centre
One core	
Peak performance	12.8 GFlop/s per socket (two cores)
Clock rate	1.6 GHz
Cores	2 per socket
Level 1 data cache	16 KByte, 64 Byte per line
Level 2 cache	256 KByte, 128 Byte per line
Level 3 cache	9 MByte, 128 Byte per line
Memory	4 GByte
Memory bandwidth	8.5 GByte/s shared between 2 cores
Cluster	
Nodes	9728 cores 19 compute partitions 256 sockets per compute partition
Type	NUMAlink 4

PowerPC	
Processor	PowerPC 450
Vendor	IBM
Bit	32
Cluster	Jugene - Jülicher Blue Gene/P
Location	Jülich Supercomputing Centre
One core	
Peak performance	13.6 Gflops/node
Clock rate	0.85 GHz
Cores	4 per node
Level 1 data cache	32 KByte per node
Level 2 cache	-
Level 3 cache	8 MByte
Memory	2 GByte
Memory bandwidth	13.6 shared between 4 cores
Cluster	
Nodes	16×1024 compute nodes
	65536 cores
Type	Three-dimensonal torus

C Notation

Symbol		*Description*
d	$d \geq 2$	Spatial dimension of continuous problem.
$\mathcal{P}$		Powerset of a set.
$\mathcal{P}_{\text{i}}$		Predicate, i.e. function to $\{true, false\}$. The text also refers to predicates as flags.

Spacetree

The following symbols, sets, and relations are used within the k-spacetree context. All of them stem from Chapter 2.

Symbol		*Description*
k	$k \geq 2$	Number of cuts along every coordinate axis.
$\mathcal{T}$		A spacetree.
$\mathbb{E}_{\mathcal{T}}$		Set of geometric elements (hypercubes) of the spacetree $\mathcal{T}$.
$\mathbb{V}_{\mathcal{T}}$		Set of vertices within the spacetree $\mathcal{T}$.
$\mathbb{H}_{\mathcal{T}}$	$\subset \mathbb{V}_{\mathcal{T}}$	Hanging vertices.
$\sqsubseteq_{\text{child}}$	$\in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}$	Partial order representing father-child relations in the spacetree. If the algorithm refines a hypercube, all the k^d resulting smaller hypercubes are children of this hypercube.
$\sqsubseteq_{\text{pre}}$	$\in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}$	Partial order on siblings, i.e. geometric elements having the same parent. The order defines which element is traversed before which sibling element.
$invert:$	$\mathcal{T} \mapsto \mathcal{T}$	Maps a spacetree to a spacetree with inverted child order $\sqsubseteq_{\text{pre}}$.
$\sqsubseteq_{\text{dfo}}$	$\in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}$	Denotes a depth-first order on the k-spacetree.
$\sqsubseteq_{\text{lw}}$	$\in \mathbb{E}_{\mathcal{T}} \times \mathbb{E}_{\mathcal{T}}$	Denotes a level-wise depth-first order on the k-spacetree.

Grid Properties and Access Operations

List of operations and properties defined on the k-spacetree entities. The implementation does not provide all of these operations, as some encode for example the complete grid connectivity and adjacency information. Not all this information however is available (all the time).

Function	*Description*
$adjacent : \mathbb{V}_{\mathcal{T}} \mapsto \mathcal{P}(\mathbb{E}_{\mathcal{T}})$	Yields the adjacent geometric elements of a vertex.
$first : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{N}_0$	Returns the moment within one traversal when an element is read for the first time.
$level : \mathbb{V}_{\mathcal{T}} \mapsto \mathbb{N}_0$	Takes a vertex and delivers the vertex's grid level equaling the level of the surrounding elements.
$\mathcal{P}_{\text{refined}} : \mathbb{E}_{\mathcal{T}} \mapsto \{\top, \bot\}$	Indicates whether an element is refined, i.e. one of the adjacent vertices holds the refined predicate.
$second : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{N}_0$	Returns the moment within one traversal when an element is read for the second time (call stack reduction)
$vertex : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{V}_{\mathcal{T}}^{2^d}$	Yields the adjacent vertices of an element.
$\wp : \partial\Omega \times \mathbb{V}_{\mathcal{T}} \mapsto \partial\Omega_{\text{h}}$	Maps a point from the continuous computational domain's boundary to one boundary vertex position along the shortest path.
$father : \ldots$	Derive the one up to 2^d father vertices, i.e. the vertices of the refined father element that influence a vertex. For complete signature see page 28.

Element Properties

The following properties are defined on each geometric element within the grid.

Property	*Description*
$level : \mathbb{E}_{\mathcal{T}} \mapsto \mathbb{N}_0$	Takes an element and delivers the element's grid level.
$\mathcal{P}_{\text{inside}}$	Element lies completely inside the discretised computational domain. In this case, invoke events on this element.
$\mathcal{P}_{\text{outside}}$	Counterpart of $\mathcal{P}_{\text{inside}}$.

Vertex Properties

The following properties are defined on each vertex within the grid.

Property	*Description*
$\mathcal{P}_{\text{boundary}}$	Vertex lies on the boundary of the discretised computational domain.
$\mathcal{P}_{\text{coarsening triggered}}$	Coarsening has been triggered for vertex, i.e. the 2^d adjacent geometric elements will be coarsened throughout the next traversal.
$\mathcal{P}_{\text{inside}}$	Vertex is inside the computational domain.
$\mathcal{P}_{\text{outside}}$	Vertex is outside of the computational domain.
$\mathcal{P}_{\text{refined}}$	Indicates whether a vertex is refined, i.e. whether all surrounding elements are refined elements.
$\mathcal{P}_{\text{refinement triggered}}$	Refinement has been triggered for vertex, i.e. the 2^d adjacent geometric elements will be refined throughout the subsequent traversal.

Automaton Properties

The following properties are defined on the stack automaton realising the Peano traversal. Most of these data are stored within the geometric elements although all values depend solely on the automaton's state in the recursion step before as well as the parent element.

Function	*Description*
$access : \{0, \ldots, 2d-1\} \mapsto \mathbb{Z}$	Attaches each element face a number. Actual range is $\{-2d+1, \ldots, 2d-1\}$. Number returns order the neighbouring elements have been visited before (negative number) or will be visited (positive number).
$even : \mathbb{E}_{\mathcal{T}} \mapsto \{\top, \bot\}^d$	Splits the elements up into odd and even elements along each coordinate axis.

Finite Elements

The finite element method uses the following symbols to discuss the solution of the Poisson equation. They define the computational domain, the function spaces holding the weak solutions, and the spacetree's discretised approximation spaces holding the numerical solution. Finally, the table gives the symbols of the operations mapping these functions to each other.

Symbol		*Description*
ℓ	$\ell \in \mathbb{N}_0$	Level within grid.
Ω	$\Omega \subset \mathbb{R}^d$	Computational domain. Bounded open subset of the d-dimensional space with sufficiently smooth boundary. The partial differential equation is defined on Ω.
$\partial\Omega$		Boundary of computational domain.
Ω_h		Discretised computational domain. Also fine grid.
$\Omega_{\mathrm{h},\ell}$		Grid on level ℓ.
$\partial\Omega_\mathrm{h}$		Boundary of fine grid.
$\Omega_{\mathrm{h},\ell}$		Grid of level ℓ.
$\Omega_{\mathrm{h},\ell}^{\text{adaptive}}$		Adaptive grid up to level ℓ.
$H^1(\Omega)$		Sobolev space for the Poisson problem.
$H^1_\mathrm{h}(\Omega_\mathrm{h})$	$\subset H^1(\Omega)$	Discretised subspace of the Sobolev space. It is spanned by a d-linear nodal basis (hat functions) on the fine grid.
$H^1_\mathrm{h}(\Omega_{\mathrm{h},\ell})$		Discretised Sobolev space spanned by the nodal basis of one grid level.
$H^1_\mathrm{h}(\Omega_{\mathrm{h},\ell}^{\text{adaptive}})$		Discretised Sobolev space spanned by the hat functions on $\Omega_{\mathrm{h},\ell}^{\text{adaptive}}$.
$H^1_\mathrm{h}(\Omega_\mathcal{T})$		Generating system on k-spacetree.
φ	$\in H^1(\Omega)$	Function from the Sobolev space.
ϕ	$\in H^1_\mathrm{h}(\Omega_\mathrm{h})$	(Hat) Function from the discretised Sobolev space. It has local support and its support covers 2^d geometric elements of one grid level.
$h : H^1(\Omega) \mapsto H^1_\mathrm{h}(\Omega_\mathcal{T})$		Takes a function from $H^1(\Omega)$ and delivers a representation within the k-spacetree.
$\hat{h} : H^1_\mathrm{h}(\Omega_\mathcal{T}) \mapsto H^1_\mathrm{h}(\Omega_\mathrm{h})$		Counterpart of h. If h's preimage is from $H^1_\mathrm{h}(\Omega_\mathrm{h})$, applying h and $\hat{h}$ in a row yields the identity, i.e. $(\hat{h}\cdot h)u = u,\ \forall u \in H^1_\mathrm{h}(\Omega_\mathrm{h})$.

Multigrid Operations

The following symbols are used for the multigrid Poisson solver. The table starts with function symbols and continues with the values assigned to the grid's vertices. It ends up with the operators and matrices applied to these values.

Symbol	*Description*
ℓ	Active level, i.e. level the smoother is currently processing.
$u_{\mathrm{h},\ell}$	Approximation of the solution on level ℓ.
$b_{\mathrm{h},\ell}$	Right-hand side on level ℓ.
$r_{\mathrm{h},\ell}$	Residual belonging to the approximation on level ℓ.
$e_{\mathrm{h},\ell}$	Error of the approximation on level ℓ.
$\hat{u}_{\mathrm{h},\ell}$	Hierarchical surplus of the approximation on level ℓ with respect to the approximation on level $\ell - 1$.
$\hat{r}_{\mathrm{h},\ell}$	Hierarchical residual resulting from $\hat{u}_{\mathrm{h},\ell}$.
u_v	Value of the current approximation at a position in space. Is assigned to each inner vertex. For boundary vertices, it determines the Dirichlet boundary condition.
$\hat{u}_v$	Hierarchical surplus of a solution stored in vertex v. Value is usually stored within vertex variable u_v.
r_v	Residual of the current approximation in vertex.
$\hat{r}_v$	Hierarchical residual at a vertex's position. Value is usually stored within vertex variable r_v.
b_v	Right-hand side of the PDE. Value is assigned to each inner vertex.
$\tilde{u}_v$	Mean value at a vertex. Results from the surrounding vertices' values u_v. $u_v - \tilde{u}_v$ gives the linear surplus.
A	Stiffness/system matrix resulting from the weak formulation of the Poisson equation with a nodal ansatz space.
P	Prolongation of a solution to the next finer level. This thesis solely applies full-weightening corresponding to the hat functions.
R	Restriction of a residual. This thesis realises a Galerkin multigrid approach, i.e. $R = P^T$.
C	Coarsening operator. This thesis uses the trivial induction, i.e. for each coarse grid vertex coinciding with a fine grid vertex, the fine vertex's value is copied to the coarse grid value.
$\mathcal{P}_{\text{unrefined } v}$	Identifies refined elements from $\mathbb{E}_{\mathcal{T}}$ where the adaptive coarse grid smoothing also evaluates the stencil.
$\|.\|_{\max}$	Maximum norm.
$\|.\|_{\mathrm{h}}$	h-dependent norm, i.e. norm that takes grid layout into account. For $h \to 0$, the norm converges to $\|.\|_{L_2}$.

Parallelisation

The following predicates and properties are introduced throughout the parallelisation chapter. They hold both the load balancing and the domain decomposition data.

Symbol		*Description*
C_{weight}	≥ 1	Weight of a leaf.
δ		Delta of a geometric element. Gives the number of additional work units the processor handling the subtree defined by this geometric element could handle without becoming a bottleneck for its master.
$FCFS$		"First come first served" answering strategy of the node pool server.
$Fair$		Fair answering strategy of the node pool server, i.e. the node pool tries to treat all computational nodes equally and sort the requests accordingly.
p	≥ 1	Number of (logical) processors available.
$\mathcal{P}_{\text{fork}}$		Fork predicate holds for a leaf if a refinement would slow down the traversal.
$\mathcal{P}_{\text{join}}$		An element is remote, but it could be merged into the master's partition. The master is a lazy master.
$\mathcal{P}_{\text{wait}}$		Wait predicate holds for a geometric element if the traversal has to wait within this element for a worker.
$rank$	$0 \leq rank < p$	Rank (number) of a processor.
$subLevel$		Holds the ranks of the processes that are responsible for the 2^d adjacent elements of the vertex at the same position in space but on the next level.
$thisLevel$		Holds the ranks of the processes that are responsible for the 2^d adjacent elements of the vertex.
w	$w \geq 0$	Weight of a geometric element, i.e. the costs of the element plus all the descendants.
w_{local}		Sum of the weights of all the local children of a refined element.
w_{remote}		Maximum of the weights of all the remote children of a refined element.

Symbol	*Description*
derive	Derives *thisRank* and *subLevel* entries for a new vertex. Uses the parent element's vertices.
mergeWithNeighbour	Merges *thisRank* and *subLevel* of a vertex with a remote vertex's lists. Each rank is allowed to modify entries it is responsible for. This happens due to merged and joins.

Grid Traversal and Storage

The following operations evaluate the traversal stack automaton. Their result is used to realise the grid management, i.e. the operations deliver information where to take vertex data from and where to write vertex data to. If a symbol accepts arguments besides the stack automaton, these arguments are mentioned in the description. The results are formulated as questions.

Symbol	*Description*
$\mathcal{P}_{\text{touched}}$	Accepts number of face. It is a proxy to evaluate *access*. Has neighbouring element connected by face been visited before, i.e. have the vertices belonging to this face been read and written by the neighbouring element?
belongsToTouchedFace	Is given the $2d$ touched predicates assigned to the faces of an element and the position of a vertex within this element. Is there one touched face adjacent to this vertex?
belongsToUntouchedFace	Mirrors *belongsToTouchedFace*. It there one untouched face adjacent to the vertex?
isPositiveAlongAxis	Accepts an axis number. Peano iterate runs along a coordinate axis?
getReadStack	Accepts a vertex position and returns the number of the temporary stack if this vertex is to be loaded from a temporary stack. Otherwise, it returns `UseInputStream`, and the algorithm has to take the vertex from the input stream.
getWriteStack	Counterpart of *getReadStack*.

Bibliography

[1] M. Bader. *Robuste, parallele Mehrgitterverfahren für die Konvektions-Diffusions-Gleichung*. Herbert Utz Verlag, Dissertation, Technische Universität München, 2001.

[2] M. Bader, S. Schraufstetter, C. A. Vigh, and J. Behrens. Memory Efficient Adaptive Mesh Generation and Implementation of Multigrid Algorithms Using Sierpinski Curves. *International Journal of Computational Science and Engineering*, 4(1):12–21, 2008.

[3] 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.

[4] 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.

[5] M. R. Benioff and E. D. Lazowska. *Report to the President. Computational Science: Ensuring America's Competitiveness*. President's Information Technology Advisory Committee, 2005.

[6] B. Bergen. *Hierarchical Hybrid Grids: Data Structures and Core Algorithms for Efficient Finite Element Simulations on Supercomputers*, volume AS14 of *Advances in Simulation*. SCS Europe, Dissertation, Friedrich-Alexander-Universität Erlangen, 2005.

[7] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[8] D. Braess. *Finite Elements—Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, 3rd edition, 2007.

[9] M. Brenk, H.-J. Bungartz, M. Mehl, I. L. Muntean, T. Neckel, and T. Weinzierl. Numerical simulation of particle transport in a drift ratchet. *SIAM Journal of Scientific Computing*, 30(6):2777–2798, 2008.

[10] W. L. Briggs, H. Van Emden, and S. F. McCormick. *A Multigrid Tutorial*. Cambridge University Press, 2nd edition, 2000.

[11] M. Broy. *Informatik—Eine grundlegende Einführung. Programmierung und Rechnerstrukturen*. Springer-Verlag, 2nd edition, 1997.

[12] M. Broy. *Informatik—Eine grundlegende Einführung. Systemstrukturen und Theoretische Informatik*. Springer-Verlag, 2nd edition, 1998.

[13] H.-J. Bungartz, W. Eckhardt, M. Mehl, and T. Weinzierl. Dastgen - A Data Structure Generator for Parallel C++ HPC Software. In M. Buback, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, editors, *ICCS 2008 Proceedings*, Lecture Notes in Computer Science, Heidelberg, Berlin, 2008. Springer-Verlag.

[14] 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*, 2009. (in press).

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

[16] H.-J. Bungartz, M. Griebel, and C. Zenger. *Einführung in die Computergraphik: Grundlagen, geometrische Modellierung, Algorithmen*. Vieweg+Teubner Verlag, 2nd edition, 2002.

[17] 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.

[18] 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.

[19] C. Burstedde, O. Ghattas, G. Stadler, T. Tu, and L. C. Wilcox. Towards Adaptive Mesh PDE Simulations on Petascale Computers. In *Proceedings of Teragrid '08*, published electronically on www.teragrid.org, 2008.

[20] M. de Berg, O. Cheong, and M. van Kreveld. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[21] N. Dieminger. Kriterien für die Selbstadaption cache-effizienter Mehrgitteralgorithmen. Diploma Thesis, Fakultät für Mathematik, Technische Universität München, 2005.

[22] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.

[23] W. Eckhardt. Automated Recursion Unrolling for a Dynamical Adaptive PDE Solver. Diploma Thesis, Fakultät für Informatik, Technische Universität München, 2009.

[24] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, 2002.

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

[26] C. Freundl, T. Gradl, and U. Rüde. Towards Adaptive Mesh PDE Simulations on Petascale Computers. In *Petascale Computing. Algorithms and Applications*, pages 375–389. Chapman & Hall/CRC Computational Science, 2008.

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

[28] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. In *IEEE Transactions on Image Processing*, volume 5, pages 794–797, 1996.

[29] T. Gradl and U. Rüde. High Performance Multigrid in Current Large Scale Parallel Computers. In *9th Workshop on Parallel Systems and Algorithms (PASA)*, volume 124, pages 37–45. GI Edition: Lecture Notes in Informatics, 2008.

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

[31] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 25(7):827–843, 1999.

[32] M. Griebel and G. W. Zumbusch. Hash-Storage Techniques for Adaptive Multilevel Solvers and their Domain Decomposition Parallelization. In J. Mandel, C. Farhat, and X.-C. Cai, editors, *Proceedings of Domain Decomposition Methods 10, DD10*, number 218, pages 279–286, 1998.

[33] Michael Griebel. *Multilevelmethoden als Iterationsverfahren über Erzeugendensystemen.* Teubner Skripten zur Numerik. Teubner, Habilitation, Technische Universität München, 1994.

[34] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. High-performacne parallel implicit CFD. *Parallel Computing*, 27(4):337–362, 2001.

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

[36] 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.

[37] D. Hackenberg, R. Schöne, W.E. Nagel, and S.Pflüger. Optimizing OpenMP Parallelized DGEMM Calls on SGI Altix 3700. In W. E. Nagel, W. V. Walter, and W. Lehner, editors, *Euro-Par*, volume 4128 of *Lecture Notes in Computer Science*, pages 145–154. Springer-Verlag, 2006.

[38] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. *Physics of Fluids*, 8(12):2182–2189, 1965.

[39] J. Hartmann. Entwicklung eines cache-optimalen Finite-Element-Verfahrens zur Lösung d-dimensionaler Probleme. Diploma Thesis, Institut für Informatik, Technische Universität München, 2004.

[40] 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.

[41] T. Huckle. Compact fourier analysis for designing multigrid methods. *SIAM Journal on Scientifc Computing*, 31(1):644–666, November 2008.

[42] J. Hungershöfer and J.-M. Wierum. On the quality of partitions based on space-filling curves. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pages 36–45. Springer-Verlag, 2002.

[43] S. Iqbal and G. F. Carey. Performance analysis of dynamic load balancing algorithms with variable number of processors. *Journal of Parallel and Distributed Computing*, 65(8):934–948, 2005.

[44] D. E. Keyes. Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.

[45] D. E. Keyes. Domain Decomposition Methods in the Mainstream of Computational Science. In *Proceedings of the 14th International Conference on Domain Decomposition Methods*, pages 79–93. Published by the National Autonomous University of Mexico (UNAM), 2003.

[46] 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.

[47] D. E. Knuth. *The Art of Computer Programming - Volumes 1–3.* Addison-Wesley Professional, 2nd edition, 1998.

[48] 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.

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

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

[51] M. Lieb. A full multigrid implementation on staggered adaptive cartesian grids for the pressure poisson equation in computational fluid dynamics. Master's thesis, Institut für Informatik, Technische Universität München, 2008.

[52] V. D. Liseikin. *Grid Generation Methods.* Springer-Verlag, 1st edition, 1999.

[53] S. Meyers. *Effective STL.* Addison-Wesley, 2001.

[54] W. F. Mitchell. A Parallel Multigrid Method Using the Full Domain Partition. In *Special Issue for Proceedings of the 8th Copper Mountain Conference on Multigrid Methods*, volume 6, pages 224–233. Electronic Transactions on Numerical Analysis, 1998.

[55] W. F. Mitchell. The Full Domain Partition Approach to Distributing Adaptive Grids. In *Proceedings of international centre for mathematical sciences on Grid adaptation in computational PDES: theory and applications*, pages 265–275. Elsevier, 1998.

[56] W. F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67(4):417–429, 2007.

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

[58] R.-P. Mundani. *Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben*. Berichte aus der Informatik. Shaker Verlag, Dissertation, Universität Stuttgart, 2006.

[59] T. Neckel. Einfache 2d-Fluid-Struktur-Wechselwirkungen mit einer cache-optimalen Finite-Element-Methode. Diploma Thesis, Fakultät für Mathematik, Technische Universität München, 2005.

[60] T. Neckel. *The PDE framework Peano: An environment for efficient flow simulations*. Verlag Dr. Hut, Dissertation, Technische Universität München, 2009.

[61] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski. A common data management infrastructure for parallel adaptive algorithms for pde solutions. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–22. ACM Press, 1997.

[62] T. Plewa, T. Linde, and V. G. Weirs. *Adaptive Mesh Refinement - Theory and Applications*. Springer-Verlag, 2005.

[63] 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.

[64] U. Rüde. *Mathematical and computational techniques for multilevel adaptive methods*, volume 13 of *Frontiers in Applied Mathematics*. SIAM, Habilitation, Technische Universität München, 1993.

[65] R. Rugina and M. C. Rinard. Recursion unrolling for divide and conquer programs. In S. P. Midkiff, J. E. Moreira, M. Gupta, S. Chatterjee, J. Ferrante, J. Prins, W. Pugh, and C.-W. Tseng, editors, *LCPC '00: Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, volume 2017 of *Lecture Notes in Computer Science*, pages 34–48. Springer-Verlag, 2001.

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

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

[68] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[69] S. Schraufstetter. Speichereffiziente Algorithmen zum Lösen partieller Differentialgleichungen auf adaptiven Dreiecksgittern. Diploma Thesis, Fakultät für Mathematik, Technische Universität München, 2006.

[70] B. Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 4th edition, 2000.

[71] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 3(30):202–210, 2005.

[72] J. F. Thompson, B. K. Soni, and N. P. Weatherill. *Handbook of Grid Generation*. CRC Press, 1998.

[73] U. Trottenberg, A. Schuller, and C. Oosterlee. *Multigrid*. Academic Press, 1st edition, 2000.

[74] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[75] T. Wagner. Randbehandlung höherer Ordnung für ein cache-optimales Finite-Element-Verfahren auf kartesischen Gittern. Diploma Thesis, Fakultät für Mathematik, Technische Universität München, 2005.

[76] T. Weinzierl. Eine cache-optimale Implementierung eines Navier-Stokes Lösers unter besonderer Berücksichtigung physikalischer Erhaltungssätze. Diploma Thesis, Institut für Informatik, Technische Universität München, 2005.

[77] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 1–31. ACM Press, 1999.

[78] I. Yavneh. Why Multigrid Methods Are So Efficient. *Computing in Science & Engineering*, 8(6):12–22, 2006.