Exploiting Many-Core Architectures for Dimensionally Adaptive Sparse Grids

Gerrit Buse
Acknowledgments

I want to express my sincere gratitude towards my supervisor Hans-Joachim Bungartz, who not only encouraged me to join his group and work on this thesis but also never ceased to support my efforts. After several years, I’m still impressed by his patience and loyalty on which one can always count.

My other scientific supervisor, Dirk Pflüger, has my sincere gratitude as well for being equally supportive, patient, and reliable. He drew my interest to sparse grids, and by keeping open the discussion between the members of the sparse grid group ensured that it never got boring.

Special thanks also goes to Riko Jacob, whose ideas made me eventually choose the final direction of my research and whose creativity and enthusiasm I came to admire.

I want to further thank the Technische Universität München and the King Abdullah University of Science and Technology for giving me the chance to work in the collaboration project Virtual Arabia, the organizers of the Ferienakademie for interesting seminars and spectacular hikes, and the people at IGSSE for offering numerous inspiring activities.

Of course, I want to thank many more colleagues and friends at TUM. This includes my coworkers from the Chair of Scientific Computing, my project partners from the Virtual Arabia project, people I have met in and through the IGSSE, but most of all the 2007 and 2008 Ferienakademie participants, who gave me another reason to come to Munich. All these people have been a constant source of ideas, support, encouragement, cookies, puns, laughter, and distraction whenever one of these was needed.

I’m also indebted to my helpful student Matthias Fischer, who directly contributed to several experiments conducted for this thesis.

Furthermore, I want to specifically thank the organizers and the developers of the SG++ project. Many of the experiments done for this thesis have greatly benefited from joint discussions or from solutions and experiments documented in the project repository.

Finally, I have probably not told them as often as I should have, but I’m deeply grateful to and for my family. They patiently left me to my work when I asked them to (or didn’t) and still supported me with everything from encouragement to delicious food.
Abstract

Sparse grids are by now widely used in simulation and approximation settings of moderately high dimensionality. Their attractiveness stems from the superior cost-benefit ratio they offer compared to other numerical discretization techniques. On the downside, bringing together the hierarchical nature of the grids and today’s many-core architectures is a tough challenge. Even state-of-the-art sparse grid codes struggle to take full advantage of multiple processors, wide vector registers, and fast caches in deep memory hierarchies, especially when adaptivity is involved.

One typical reason are the data structures used. The most popular ones for instance, hash maps, provide a convenient interface to the grids’ complex structure but come with obvious shortcomings: lookups are costly, stored data is typically scattered and cannot be easily aligned for vectorized processing. As a consequence, solutions relying on hash maps have not seen any big performance leaps lately. Yet, attempts are still made to finally lift sparse grids to the era of real-time computing. Recent approaches abandon sophisticated algorithms and data structures, trading them against less optimized forms drawing benefit from massive parallelism on large clusters.

This work follows a different idea. The focus is on dimensionally adaptive sparse grids, a special type of adaptive grids sharing a structural resemblance with classical regular sparse grids. For these grids, a novel set of co-designed data structures and algorithms is presented and shown to meet the performance requirements mentioned. The potential is manifold. This is substantiated by excellent performance results achieved on multiple platforms. Thus, real-time sparse grid applications become a practical possibility even without the need for large clusters. Furthermore, the developed solutions incorporate software abstractions to hide away generally applicable low-level optimizations. The presented design helps to exploit the system’s computational power, while relieving developers to a large extent from the responsibility to explicitly account for the computer’s performance-critical resources.
Zusammenfassung


Contents

1. Introduction 1

2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms 7
   2.1. Many-Core Systems 8
      2.1.1. Walls Everywhere – Recent Challenges in Hardware Design 8
      2.1.2. Parallelism! 11
      2.1.3. Multi-Platform, Multi-Layer, Multi-Paradigm 13
      2.1.4. Target Platforms and Their Ecosystems 15
   2.2. A Pragmatic Introduction to Sparse Grids 21
      2.2.1. Point Set Construction 23
      2.2.2. Sparse Grid Theory and Notation 25
      2.2.3. Hierarchical Bases 31
      2.2.4. Structural Variants 36
   2.3. Sparse Grid Software – An Overview of Publicly Available Tools and Libraries 41
      2.3.1. SG++ 41
      2.3.2. Tasmanian, PECOS & Co. 42
      2.3.3. fastsg and sgbench 43
      2.3.4. SPINTERP 44

3. Co-Design, Part I: Formulating the Challenges 45
   3.1. Algorithmic Implications of the Sparse Grid Construction 46
      3.1.1. Multi-Recursive Tree Structure 46
      3.1.2. A Benefit of Tensor Products: One-Dimensional Schemes 47
      3.1.3. Multi-Pass Processing and Changing Data Dependencies 48
   3.2. The Core Tasks 48
      3.2.1. Basis Transformations 49
      3.2.2. Evaluation 55
      3.2.3. UpDown – The Unidirectional Principle 57
      3.2.4. Summary of Observations 65
   3.3. Commonly Used Sparse Grid Data Structures 65
      3.3.1. Tree-Like Structures 66
      3.3.2. Hash Maps for Spatial Adaptivity 69
      3.3.3. Array of Structs (AoS) and Struct of Arrays (SoA) 71
      3.3.4. Hash Maps for Dimensional Adaptivity 76
### Contents

3.3.5. Bijective Mappings ........................................... 77

4. Co-Design, Part II: Presenting Solutions ................................ 81
   4.1. Towards Optimized Code for Dimensionally Adaptive Sparse Grids ... 81
   4.2. A Proactive Data Structure for Dimensionally Adaptive Sparse Grids .... 82
      4.2.1. Separation of Data and Structure: Ensuring a Versatile Design .... 83
      4.2.2. Iterator-Based Traversal: Becoming Part of the Algorithm ......... 86
      4.2.3. Task Awareness: Anticipating Data Access .......................... 88
   4.3. The Core Tasks Solved ...................................... 95
      4.3.1. Basis Transformations ..................................... 95
      4.3.2. Evaluation ............................................. 103
      4.3.3. UpDown ............................................... 112
      4.3.4. Summary of Results .................................... 120
   4.4. Software Design .......................................... 121
      4.4.1. Exchange with the SG++ Project .......................... 122
      4.4.2. Class Structure and Extension Points .......................... 123

5. Dimensionally Adaptive Sparse Grids in Action .......................... 127
   5.1. Sparse Grids for Classification .................................. 128
      5.1.1. A Regularized Least Squares Approach for Classification ....... 128
      5.1.2. Solution of the System of Linear Equations (SLE) ................ 129
      5.1.3. Results ............................................. 130
   5.2. Sparse Grids for Out-of-Core Visualization ......................... 133
      5.2.1. Out-of-Core Rendering and the Role of Wavelets ............... 133
      5.2.2. Computational Steering via Surrogate Models: Chances and Challenges ............. 135
      5.2.3. The Hierarchical Transform on Full Grids .................... 139
      5.2.4. Results ............................................. 144
   5.3. Sparse Grids for Solving Partial Differential Equations ............ 148
      5.3.1. Application: Option Pricing ................................ 148
      5.3.2. The Black-Scholes Equation ................................ 150
      5.3.3. Finite Elements on Sparse Grids for Parabolic PDEs ............ 152
      5.3.4. Concurrent Processing of the UpDown Tree .................... 154
      5.3.5. Results ............................................. 156
   5.4. Applying the Concepts to Spatially Adaptive Sparse Grids ........... 162
      5.4.1. The Key-Value Approach Briefly Revisited ................... 163
      5.4.2. Elimination of the Sorting Step ................................ 164
      5.4.3. Vectorization in an Adaptive Setting .......................... 168
      5.4.4. Results ............................................. 169

6. Conclusions ................................................................ 173

A. Formulations of Selected 1-D Operators .................................. 177
   A.1. Linear Hierarchization ........................................ 177
A.2. The Prewavelet Transform ................................................. 178
B. Hierarchical Transforms for Out-of-Core Visualization .......... 181
B.1. Construction of Orthonormal Wavelets ............................... 181
B.2. The Multi-Level Subspace Splitting for Reconstruction of Image Data . 182
C. Solution of the Black-Scholes Equation for Option Pricing .... 185
C.1. Transformation into the Heat Equation ................................. 185
C.2. Documentation of Simulation Parameters .............................. 186
1. Introduction

*Everyone knows Amdahl’s law, but quickly forgets it.*

— Dr. Thomas Puzak, IBM, 2007 [72]

Now, for those who have forgotten, here is a quick reminder. In his 1967 article [5], Gene M. Amdahl analyzes the potential of the “multiple processor approach” and its “application to real problems and their attendant irregularities”. One of his statements later becomes famous as *Amdahl’s Law*. An excerpt from the corresponding paragraph reads as follows:

> [...] The first characteristic of interest is the fraction of the computational load which is associated with data management housekeeping. [...] The nature of this overhead appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. [...] A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. [...] 

From these sentences, the famous formula for a program’s expected parallel speedups

\[
S(n) := \frac{T(1)}{T(n)} = \frac{1}{r_{seq} + r_{par}/n},
\]

is derived, where \(T(n)\) is the execution time on \(n\) parallel processors, \(r_{seq}\) is the fraction of the application’s sequential “data management housekeeping” overhead, and \(r_{par} = 1 - r_{seq}\) is the (infinitely) parallelizable fraction of the program.

The picture Amdahl paints for the “multiple processor approach” is a dark one. He refers to an application’s strong scaling, which is hard to achieve in practice, also because the perfect scheduling he assumes is virtually impossible. Yet, sometimes it is good enough to just maintain the same time to solution when tackling a scaled-size problem on a larger number of nodes (see *Gustafson’s Law* [62]). This corresponds to an application’s weak scaling and is often much easier accomplished. The problem with the sparse grids approach I consider in this thesis is that it does not fit well with distributed processing. Hence, the prime challenge for an efficient sparse grid implementation in the multi-core era is to make effective use of a system’s parallel resources in order to beat the odds described by Amdahl. In the following, I will briefly explain where we stand before and after this thesis.
First, what are sparse grids? The name refers to a discretization technique first used by Zenger in 1991 [132] for solving partial differential equations (PDEs). The grids described by Zenger use a truncated hierarchical data representation to retain a high storage efficiency also in settings of \( d \geq 4 \) dimensions. Standard grid-based approaches usually fail here, as they require \( O(N^d) \) sampling points in order to maintain a prescribed accuracy in numerical computations, where \( N \) is the number of sampling points used to discretize a single dimension. Because of this unique property and their general purpose character, sparse grids have meanwhile been applied to many other tasks besides solving PDEs, e.g., quadrature, regression, classification, stochastic collocation, uncertainty quantification, and computational steering, to name just a few. Furthermore, the continuously growing international user community has started to hold a bi-annual summit, the “Sparse Grids and Applications” workshop.

Obviously, sparse grids are very relevant. And yet, browsing through publications reveals a puzzling gap: While more and more scientists from various computational disciplines have jumped on the sparse grid train, the line of research related to sparse grid data structures has almost come to a halt. It is a fact confirmed by the literature study conducted for this thesis that merely a single digit number of publications containing innovative ideas regarding sparse grid data structures date from years after 2000. And out of these “recent” solutions, more than half fail to address the matter of parallelism, simply because they date back to days before chips went multi-core.

A large part of the sparse grid community will now argue that this is not true or does not matter, since the sparse grid combination technique (CT) invented by Griebel and Zenger in 1992 [59] is an extremely vivid line of research and is furthermore inherently parallel. The CT is even generally considered the sparse grid technique’s only chance of making it to the exascale [108]. All of this is certainly true, however, this thesis is about the direct sparse grid approach and not about the CT. Accordingly, those people from the sparse grid community who do not use the CT will now either be surprised by this small number of publications, or they are already aware of the problem.

The ugly truth is, the price for the extreme efficiency of the sparse grid sampling is high, and it seems to be increasing in the era of multi-core processors. In the hierarchical basis used with sparse grids, even standard tasks often lead to complex algorithms, whose steps are governed by data dependencies within a multi-dimensional hierarchy of grid points. As a result, both programmers and computers are challenged, and optimal solutions for both sides are difficult to find. The problems with the hash map, the dominating sparse grid data structure since the late 1990s, are symptomatic: The hash map conveniently takes care of data organization for the programmer, but it does so ignorant of specific sparse-grid-related tasks. Resources such as fast caches or vector registers are thus left completely unused, even though they ultimately define the performance characteristics of today’s compute platforms. Multi-threading suffers, too, as the hash map layer disallows efficient synchronized data access.

Nevertheless, two recently published approaches show that we can do better. Both
approaches represent successes and are instructive, however, they also have limitations.

- A streaming-based approach to spatially adaptive sparse grids reaches machine peak performance on several parallel architectures, thus permitting to solve a five-dimensional option pricing problem on 512 cluster nodes in record time [67, 66]. However, Gustafson’s Law only partially applies to this example, as no scaling of the problem size takes place. Instead, higher workload for a fixed-size problem is purposefully generated through the deliberate choice of suboptimal $O(N^2)$ schemes. Regarding time to solution, the available runtime-optimal $O(N)$ schemes are still beaten, as they do not allow for embarrassingly parallel cluster implementations. On the downside, the suboptimal schemes do not scale to larger problem sizes.

- In a series of publications about regular sparse grids [94, 93, 24], it is shown that also Amdahl’s odds can successfully be fought. On shared memory systems, the related implementations achieve remarkable strong scaling results for the hierarchization algorithm, a basis transformation serving as representative of a class of more complex algorithms. In contrast to fully adaptive sparse grids, the structure of regular grids is a priori known. This facilitates vectorization and improved parallel load balancing, but it limits the grids’ general applicability.

Motivated especially by the results of the second approach, the first purpose of this thesis is to demonstrate that highly efficient sparse grid implementations can exploit the resources of modern parallel (non-distributed) platforms and defy Amdahl’s odds, even if the grids’ structure is not a priori known. The targeted grid type are dimensionally adaptive sparse grids (DASG) as first proposed by Hegland [64]. DASG are more flexible than regular sparse grids, as they admit anisotropic samplings, but they are more restrictive than spatially adaptive sparse grids which support local refinement. The relevance of DASG is indirectly given by the widely used combination technique, as both techniques in principle solve in the same approximation spaces. Actually, however, only DASG solve directly in the sparse grid space, while the CT relies on extrapolated solutions and is known to struggle in some cases (e.g., see [43, 44]).

The second purpose of this thesis is to promote the benefits of the careful co-design of data structures and algorithms. There can be no universal handbook for this highly interwoven co-design process, nevertheless, the systematic approach taken in this thesis has lead to success and can thus serve as an instructive example for other settings as well. This claim is substantiated not only by the excellent benchmark results achieved by the DASG implementation. The design principles established for and applied in the DASG components’ design process were in addition used to guide the implementation of a novel high performance data structure for spatially adaptive sparse grids.

The third purpose of this thesis is to demonstrate that carefully designed software interfaces also have their gain in scientific computing. Software abstractions are used to facilitate the development of hardware-aware high performance implementations within
1. Introduction

the shortest time. To this end, specifics of the platform are hidden in efficient compute kernels that ensure optimal use of the hardware’s performance-critical resources such as caches, hardware prefetchers, and vector registers. A vast collection of mathematical operators defined for several different types of ansatz functions all meet the requirements of scientific high performance software and bear witness to an increased programmer productivity and the concepts’ success.

Since the direct sparse grid approach implies dealing with hierarchical data (structures), the main target platform in this thesis are x86 multi-core CPUs. Still, with the Intel Xeon Phi and an Nvidia Kepler GPU two cutting-edge accelerators make appearances in settings where their strengths can be leveraged. It is noteworthy, that the porting of the hierarchical data structures for DASG to the accelerators only required minor changes to the general schemes. This is again proof of flexible software constructs that are well-adapted to the requirements of state-of-the-art parallel computing devices.

A more detailed list of the contents of this thesis is given in the following. Alongside the short chapter summaries, Figure 1.1 can be used to understand the general structure. Consistent coloring in the depiction groups different sections that are closely related.

Chapter 2: First, an overview of the current landscape of parallel computing devices and their ecosystems is given, followed by the description of the three considered target platforms. Afterwards, the construction, underlying idea, and various forms of sparse grids are addressed. The chapter is concluded with an overview and the first ever assessment of publicly available sparse grid software.

Chapter 3: This chapter is the first of two chapters about the co-design of algorithms and data structures. Together, both chapters form the core of this thesis. Chapter 3 contains an extensive analysis of sparse grid theory, algorithms, and available data structures. The aim of this analysis is to extract general constraints, data patterns, and data dependencies, but also to identify strong and weak points in existing approaches. As a part of the analysis, the core algorithms are given in clear and original form together with detailed explanations. In addition, an overview of different published sparse grid data structures is shown and discussed at length.

Chapter 4: In the second chapter related to co-design, a set of design principles is first established on the basis of the analysis in Chapt. 3. It is followed by the definition of a contemporary data structure for DASG. In three self-contained sections, solutions for the completion of the most common sparse grid tasks are presented and put to test in various benchmarks. The outcome of the performance experiments assures that the new software components indeed manage to exploit the parallelism of today’s computing platforms. At the same time, the results prove the success of the co-design approach used to develop these components. In the final part of the chapter, remarks are made about how the currently largest sparse grid toolkit SG++ influenced the software design, and how the SG++ project can in turn benefit from solutions presented in this thesis.

http://www5.in.tum.de/SGpp/
In addition, the developed software’s general design and the proper use of interfaces are explained.

**Chapter 5:** The application chapter is intended to complete the picture and help readers quantify the relevance, impact, and versatility of the developed software. In the first three applications, the competitiveness and potential of the DASG implementation is demonstrated. A fourth application stands as proof for a broader applicability of the presented concepts.

The first application is to be considered an extended benchmark that tests the DASG solutions’ performance in a relevant setting. To this end, a DASG classifier competes against the currently fastest sparse grid classifier in a performance comparison.

Second, the hierarchical sparse grid transform is reinterpreted as a wavelet-like transform, in order to motivate its use in an interactive computational steering setting in need of out-of-core solutions. The GPU implementation of the transform is shown to satisfy the performance requirements of real-time visualization applications, which is the first time for an implementation based on the direct sparse grid approach.

Third, the Black-Scholes PDE for the numerical pricing of options is solved on DASG to determine the price of European basket call options in 2–7 dimensions. The simulation runtimes and accuracies achieved on a single CPU node are very close to the results obtained for spatially adaptive sparse grids on a cluster of 512 nodes.

Last, the design principles established for the DASG software are again applied, this time to define a novel data structure for spatially adaptive sparse grids. Although this small project seems at first unrelated to the rest of this thesis, its outcome is remarkable and perfectly underlines two claims of this work:

- *Applicability of the approach:*  
  The co-design approach and the derived concepts yield a broader applicability and are by no means limited to the primarily targeted DASG implementation.

- *Usability of the software:*  
  The designed software interfaces facilitate the rapid development of high performance software components and allow the flexible integration of new functionality.

**Chapter 6:** The contributions of this thesis are summarized, and remarks are made on possible future work.
1. Introduction

Figure 1.1.: The structure of this thesis is sketched. Colors connect different parts that are closely related. The inclusion in Chapt. indicates that the single parts of the analysis build upon each other.
2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

The age of parallel computing platforms poses great challenges for software developers. Sequential legacy codes, unfitting programming metaphors, and receding processor clock rates inhibit performance on systems that boast with large peak FLOP rates on the one hand, but on the other hand offer little chance of ever reaching these marks in real applications due to strict hardware-imposed design constraints for parallel software.

In the first section of this chapter, a look at the recent history of this current set of compute devices helps to understand why even highly optimized scientific codes struggle under the circumstances. An introduction to sparse grids follows, in which special attention is given to hardware constraints and to structural properties of the grids. The conclusions drawn in this part serve as a basis for the subsequent assessment of currently available sparse grid codes with regard to their fitness for the parallel platforms of today.

2.1. Many-Core Systems

The term “many-core” still lacks a clear definition and yields ambiguities. In 2006, the earlier days of the multi-core age, a group of renowned Berkeley researchers for instance use the term in a predictive fashion, referring to systems of up to around 1000 cores per chip. “Multi-core” systems on the other hand are typically equipped with 2, 4, or 8 cores at the time. In this thesis, “many-core” shall loosely stand for the full range of programmable multi-core and accelerator platforms suitable for scientific high performance computing. To represent this large variety of computing devices, I pick three target platforms that perfectly symbolize three different stages in the development of the many-core culture: an x86 architecture representing classical multi-core CPUs, a GPGPU as representative of the origins of accelerators, and a hybrid coprocessor signifying the convergence of parallel architectures currently taking place.

This section starts with a glimpse at the recent history of hardware development. I explain why the many-core culture is the inevitable outcome of a troublesome period for hardware architects, in which physical boundaries greatly determine the architects’ scope of action. I then address the levels of parallelism exhibited by these platforms and discuss which options software developers have for proper exploitation of parallel
resources. I conclude the discussion of hardware with a description of the three target platforms for which I developed software solutions in the context of this work.

2.1.1. Walls Everywhere – Recent Challenges in Hardware Design

In the early 2000s, computer hardware architects started hitting one wall after another on their mission to develop ever faster processors [104]. Since then, none of these walls have actually been overcome or torn down. Instead, the walls have continued to form obstacles during these last fifteen years, which has caused processor architects to find themselves at an impasse several times. As described in the following, a complete change of direction was necessary each time to keep processor development from stagnating, while the liberating blow taking down the walls is yet to come.

The first wall to be hit was the power wall. In times when transistors were still more expensive than power, turning up the processor frequency was the obvious thing to do. Still, in order to make this approach sustainable, the power consumption needed to be controlled. This was achieved by shrinking die sizes and shortening on-chip paths to reduce voltages. Fifty years ago, fifteen years ago, and even today, the pace of this development still matches Intel co-founder Gordon E. Moore’s 1965 prognosis known as Moore’s Law, in which he predicts a doubling of on-chip transistor density every two years. CPU frequency, however, has meanwhile peaked at around 4 GHz and then gone back to below 3 GHz for reasons of power consumption, overheating, and interfering leakage currents. This step was overdue, especially with compute clusters gaining momentum: Ensuring fail-safety for hot, power-hungry processors calls for complex cooling systems, which in turn add immensely to the electricity bill. It was clear that by relying on shrinking production processes alone, the power problem would not be solved and processor performance increases would also lag far behind historical marks. Therefore, hardware manufacturers were forced to rethink their strategy and started multiplying processor performance by multiplying the processor count. This parallelization and distribution of computing resources immediately solved the problem of overheating and reflected perfectly the spirit of an arising green computing trend: The power consumption scales linearly in the number of processors but its dependence on the CPU frequency is quadratic. Several processors can thus run highly power-efficient at low clock rates and still deliver more accumulated computing power than a single fast but power-hungry processor. Moreover, parallelism is inherent in many use cases which require that various independent tasks are processed concurrently. Examples for systems relying on this kind of task parallelism are database servers handling queries, or desktop computers running different applications in parallel.

On the one hand, the heralding of the multi-core era proved to be a way around the imminent power wall. On the other hand, chip manufacturers now steered hard for the

\[1\] Once adopted by hardware manufacturers it became a self-fulfilling prophecy, guiding the planning of resources and production processes.
2.1. Many-Core Systems

<table>
<thead>
<tr>
<th></th>
<th>1980 VAX–11/750</th>
<th>2014 Intel</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock speed (MHz)</td>
<td>6</td>
<td>3,000</td>
<td>+500x</td>
</tr>
<tr>
<td>Memory latency (ns)</td>
<td>225</td>
<td>~70</td>
<td>+3x</td>
</tr>
<tr>
<td>Mem. bandwidth (MB/s)</td>
<td>13</td>
<td>20,000 (read)</td>
<td>+1500x</td>
</tr>
<tr>
<td>FP multiplication (cycles)</td>
<td>13.5</td>
<td>0.25–4(^a)</td>
<td>+13.5x</td>
</tr>
</tbody>
</table>

Table 2.1.: The development of processor speed, memory latency, and memory bandwidth is compared for computers from 1980 and from 2014. The numbers are taken from [124].

\(^a\)The variation comes from pipelining; to simplify subsequent calculations a value of 1 is assumed.

second wall, the **memory wall**, and it soon became clear just how close this wall had already been. The problem lies in the advances memory technology has made since the introduction of dynamic random access memory (DRAM). These advances have been extremely small, especially compared to the leaps processor performance has taken meanwhile. Two examples based on the numbers in Tab. 2.1 illustrate how severely the memory wall can affect program performance: Processor speed has increased by a factor of 500 since 1980; memory latency has merely been reduced by a factor of 3 over the same time. Measured in processor cycles, memory latency is thus 150 times higher today while the latency of arithmetic operations is more than 13 times lower. According to these numbers, hiding memory access via computation became \(150 \times 13 \approx 2000\) times harder. One can think of latency as the length of a pipe through which a piece of data is retrieved. Unfortunately, there is no way to change that length. But it is common for a piece of data, just like for other work pieces, that it requires drilling from several sides or angles before it can be put aside again. Complex cache hierarchies therefore temporarily buffer hot data close to the processor and help overcome the memory latency problem to some extent. However, the second big memory-related problem of today is already waiting around the corner: memory bandwidth boundedness, i.e., the matter of the pipe’s limited width. It becomes important when the buffer is filled with fresh data and the question arises how much can be loaded at once. Consider the following scenario: A 2.5 GHz single core processor takes exactly one second to sum up the \(2.5 \times 10^9\) elements in a 20 GB double precision floating point vector. Theoretically, a modern desktop computer with 4 cores and 256 bit wide vector units provides the computing resources to reduce the computation time by a factor of 16 compared to the sequential case. Practically, the execution time will remain the same, because 20 GB/s is as wide as the “pipe” gets on such desktop systems.

The third wall and the one most recently hit is the **ILP wall**. Instruction level parallelism (ILP) has served as another way to keep processors’ performance advances at a steady level. One can think of low-level software optimizations that are mapped directly onto the hardware. Beneath all other ILP techniques instruction pipelines are employed.

\(^2\)A more expensive server platform might offer about 2.5 times more bandwidth, but such systems typically yield 2–2.5 times more cores, too.
The pipelines cascade instructions whose execution takes several processor cycles, given the single execution stages of the instructions do not allocate the same resources in the microprocessor. A filled pipeline can thus maintain an average of one retired instruction per cycle, even when multi-cycle instructions are involved. Other ILP techniques aim at keeping the pipeline filled. Out-of-order execution engines for instance reorder retired instructions dynamically without violating the data dependencies formulated in programs. Branch prediction logic (possibly enhanced with speculative execution) also tries to avoid pipeline stalls. It collects information about conditional branches taken at program runtime and applies the information when the same conditionals are evaluated again. Speculative logic sometimes allows the processor to commence simultaneous execution of conditional branches, and once the program’s actual path is determined, execution can smoothly continue without a suffered pipeline stall or flush. The above ILP techniques have proven extremely effective in preserving a sustained performance growth in processors, but the air gets thin now. It seems that for common RISC architectures (reduced instruction set computing) the possibilities are close to exhausted – at least those that come at reasonable cost. Other ILP techniques, such as very long instruction words (VLIW), rely on their own specialized microprocessor and are not suited as extensions to RISC architectures. However, VLIW architectures have so far failed to establish a notable position in the market of computing devices, the main reason being the complexity of compilers necessary to unleash the architectures’ full potential. A completely different approach meanwhile pursued by hardware developers is to induce parallelism not only between but also within instructions. To this end, vector units have been introduced as another cheap solution for boosting parallelism on the instruction level. They allow the simultaneous application of uniform operations on integer or floating point operands aligned in vector registers, and they keep growing in size, which is especially visible in accelerator architectures. In contrast to the ILP techniques discussed above, successful use of vector units relies heavily on the programmer’s formulation of a problem and works best for data parallel tasks (as opposed to the aforementioned task parallelism).

I conclude this discussion by claiming that the wall metaphor used to describe the obstacles faced in recent computer hardware development is a fitting one. It seems that the maze of indestructible walls through which hardware manufacturers had to feel their way in the last fifteen years has but one exit: parallelism. First it was only the market of computing platforms, but by now the multi-core trend has also taken over the ever-growing market of embedded systems. It is therefore time to look closer at the shape parallelism takes in these systems and discuss which means programmers have to exploit it properly.
2.1. Many-Core Systems

<table>
<thead>
<tr>
<th>single data</th>
<th>single instruction</th>
<th>multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>SIMD</td>
<td>MISD</td>
</tr>
<tr>
<td>MIMD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2.: Flynn’s taxonomy classifies computer architectures based on how many programs (instructions) can be applied to how many streams (data).

2.1.2. Parallelism!

Very obviously, there is a discrepancy between the number of available parallel compute platforms and the number of programming models aiding with the development of tailor-made parallel code. The former outweighs the latter by far. This observation is not necessarily a contradiction, it is in fact proof of a landscape of parallel platforms which is growing increasingly homogeneous, and in which many individuals already share the same software ecosystem. I conclude from this observation that a discussion of exploitable parallelism in today’s many-core architectures can be restricted to discussing only the characteristics of representative mainstream products. First, a distinction between several notions of parallelism becomes necessary. This classification then helps to identify those parallel resources exposed by the hardware, that are best suited for particular tasks.

- Parallel algorithms are commonly classified as either task parallel or data parallel. In many cases, it is hard to draw a clear line between the two. As the name suggests, data parallelism yields a more data-centric view, e.g., an implementation is typically called data parallel if it divides the data for simultaneous processing on several processors via the same program. Task parallelism is more general and also known as functional parallelism or control parallelism. In task parallel programs, assumptions are neither made about the distribution of data, nor about the subprograms executed by the processors.

- Since 1966 Flynn’s taxonomy (see Tab. 2.2) serves as a classification model for architectures with different parallel capabilities. When forming matches between programs and architectures, data parallel implementations fit best onto SIMD (also MIMD) architectures. Task parallel programs generally require MIMD architectures.

- Contemporary parallel hardware exhibits parallelism on three levels. Instruction level parallelism (ILP) is deeply buried inside the processor (cf. discussion of ILP techniques in Sect. 2.1.1), for instance in the form of vector processing units which represent classical SIMD devices. Thread level parallelism (TLP) is observed between different cores functioning as MIMD devices within a shared memory do-

---

3Some definitions consider a third type of “competitive” parallelism. The term describes a scenario in which concurrency is used to obtain a solution to a problem as fast as possible using different algorithms. In HPC this scenario is highly unlikely, and the definition is therefore ignored here.
2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

Node level parallelism (NLP) typically denotes MIMD parallelism between cores across different shared memory domains, e.g., parallelism between several network computers.

The latter differentiation is the most pragmatic one, as it directly gives us hints how to best exploit the parallel capabilities of a particular hardware in our programs. It is also the most complex one due to the large range of parallel devices placed at our disposal. The following paragraphs contain a more detailed analysis of these three levels of parallelism, illustrated at the example of the most common architecture nowadays: The x86 architecture is produced and distributed by the two largest processor manufacturers Intel and AMD, and it is present in the vast majority of notebooks, home and office computers. It has also invaded the supercomputing market and holds by now the largest share of general purpose CPUs installed in large clusters.

As mentioned before, ILP techniques cascade multi-cycle instructions in order to hide the instructions’ latency and maintain an average of one retired instruction per cycle. The last bit of performance can, however, only be squeezed out of a processor through super-scalar parallelism (SSP) – which is not easily achieved, as it is subject to many restrictions. SSP generally relies on instructions that can be processed concurrently in different execution units (e.g., in the integer unit and the floating point unit). It is up to compilers and out-of-order execution engines to identify and schedule such instructions. In contrast, vectorized processing can be considered a programmable form of SSP, but due to memory bandwidth and data alignment constraints it is not easily achieved either. It is important to emphasize at this point, that although SSP is certainly desirable, programmers’ top priority should still be to secure a high degree of general ILP. Consider the following simple calculation: Repeatedly flushing a pipeline of five stages means at least five cycles of idling for the execution units every time, which can have a heavy impact on the overall program performance. Every programmer should therefore keep in mind that the careless provocation of pipeline flushes, for instance through excessive use of conditional statements, is a performance killer.

With TLP, programmers encounter much more flexibility – but they also carry much more responsibility. The use of threading itself is not subject to any limitations from the hardware side. Yet, incorrect handling inevitably goes at the expense of program efficiency. The following list contains common guidelines and pitfalls.

- Parallelization on the thread level introduces the need for load balancing and synchronization. Section 2.1.3 presents several tools taking at least some of the complexity out of these tasks, e.g., semaphores and locks often need not be handled explicitly. Still, a lot is up to programmers and their understanding of the problems they solve.

- Threads compete for resources, which leads to race conditions. And with the memory wall towering over us, memory bandwidth is an especially scarce resource. Besides obvious resource contention problems these considerations must also include...
cache protocols. *False sharing* occurs when different cores alter neighboring memory locations and cache coherency is expensively enforced in all caches. Minimizing and localizing resource usage helps reduce the effects of such race conditions.

- Simultaneous multi-threading (SMT) should only be used to hide latency. In SMT architectures each physical compute core contains several threading contexts, so-called *logical cores*, which can execute different programs in a pseudo-parallel fashion. Logical cores share a physical core’s execution units but may duplicate resources, e.g., the register file, to allow for fast context switches. Typically, implementations only draw benefit from SMT if context switches help to hide memory latency.\(^4\) This is often achieved on massively parallel accelerators, as context switches are extremely efficient on these platforms. If an implementation does not suffer from latency boundedness, SMT tends to hinder performance by fomenting the competition for shared resources.

- Optimal performance on NUMA (non-uniform memory access) systems requires perfect knowledge about where data resides in memory. Storing data in a remote NUMA node’s local memory can help to increase the memory bandwidth an application experiences, but due to higher latency repeated access soon gets expensive. Taking advantage of NLP shifts the challenges for programmers again. On the one hand, scarce resources like memory bandwidth are multiplied when using several machines. This helps to relax the situation between processes. On the other hand, communication patterns change from shared memory mechanisms to general message passing. Network performance and reliability now play a role in all considerations. Although NLP is fundamental to all forms of cluster computing, the overview of available tools given in Sect. 2.1.3 indicates that developers aiming for NLP are often left to their own devices.

### 2.1.3. Multi-Platform, Multi-Layer, Multi-Paradigm

In the previous section’s discussion, the analysis is restricted to the x86 architecture. This makes sense as this complex general purpose architecture combines many of the performance-boosting features also employed in other state-of-the-art systems. Accelerators in comparison are often just simpler, massively parallel NUMA devices leaving out costly features such as out-of-order execution engines, branch prediction logic, and parts of the cache hierarchy. Many platforms even share the same software ecosystem, however, one must be careful not to oversimplify things. There is still an abundance of *parallel programming paradigms* about which programmers have to have good knowledge. Some of these paradigms simply coexist and provide programmers with options for their implementations. But sometimes such options become necessary choices, especially when a manufacturer-provided paradigm is the only chance to exploit special platform features.

\(^4\)Only in rare cases when long strings of dependent arithmetic operations cause an empty instruction pipeline, SMT also helps to hide the latency of multi-cycle instructions.
2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

Figure 2.1.: Parallel programming paradigms can be loosely aligned in a matrix to indicate their mutual relationships. The vertical dimension represents the different layers in parallel platforms. The horizontal dimension stands 1) for general options developers have, and 2) for necessary choices developers sometimes have to make when programming for particular platforms.

Other paradigms are orthogonal to each other and address different features in the hierarchy of parallelism. This relationship is illustrated in Fig. [2.1] where a selection of commonly used parallel programming paradigms is aligned in a two-dimensional matrix. In the following, a brief evaluation is given of the general options programmers have to draw benefit from their parallel devices.

- **Software libraries** are quite popular in scientific computing, especially when standard problems are to be solved. External modules then offer programmers the chance to exploit all levels of parallelism and experiment with different parallel hardware, without being bothered by ever-shifting programming paradigms. Nevertheless, code modularity often falls victim to the promise of the slightest performance gain. After all, library code cannot directly make use of domain specific knowledge in the efficient completion of a task.

- Fully-fledged programming paradigms manifest themselves in **programming languages**. Indeed, the general performance of code written in a particular language heavily depends on the compiler. And yet, the language itself allows to abstract over objects, patterns, behavior, and also hardware, i.e., its capabilities go beyond mere performance. C++ and Fortran are still the most widespread languages in HPC despite their conceptual shortcomings. Both languages completely lack support for TLP$^5$ and NLP, and it is in both cases the compiler that introduces ILP and TLP through auto-vectorization and auto-parallelization, respectively (possi-

---

$^5$Threading support was finally approved in the C++11 standard, but not before 2011.
The advent of specifically designed parallel domain languages such as OpenCL [91] and CUDA [100] finally proved the existence of better metaphors for modeling parallelism. These programming models pioneered with a consistent description of SIMD and task parallelism, and they have gained momentum ever since their introduction in 2007 and 2008. PGAS (Partitioned Global Address Space) languages aim at a consistent description of NLP by emulating a large NUMA-like shared memory system. They are typical cluster languages and thus bear little relevance for this thesis.

- **Compilers** obviously play a big role for parallel performance. Auto-vectorizers help exploit SIMD parallelism, their capabilities are, however, still limited. Sometimes the vectorizer accepts additional clues in form of compiler-specific pragmas or builtins. The OpenMP standard [14] defines a set of pragmas to supplement languages with TLP features (e.g., C++ and Fortran). OpenMP is nowadays supported by almost every noteworthy compiler and can be counted as portable solution. OpenACC [88] pursues a similar pragma-based approach but it goes further and aims at convenient use of accelerators as offload devices. So far the OpenACC standard is only supported by some of the committee members’ compiler solutions (e.g., Cray and CAPS). To summarize, pragma-based parallelism can be extremely effective, and at the same time it can completely inhibit performance portability.

- **Low-level APIs** (as opposed to libraries for sophisticated numerical subroutines) are typically optimized for performance and control, not for convenience. They expose most of the complexity in the interface, and their target group are experienced programmers. On the register level, SIMD parallelism on CPU-like platforms is best achieved through dedicated SSE and AVX intrinsics (not counting assembler code as a viable option). On the node level, the MPI (Message Passing Interface) library [40] is the preferred solution for communication in distributed systems.

## 2.1.4. Target Platforms and Their Ecosystems

I make a choice of three platforms for which I present software solutions in the course of this thesis. The platforms are representatives of different conceptual classes of many-core systems. The primary target platform is a general purpose CPU system with two eight-core Intel server CPUs based on the x86 architecture. Most time has been spent in the developing of new algorithms and data structures optimized for this platform. As the most flexible of the three systems, it is the only system for which all algorithms have been implemented. The second target platform is an Nvidia general purpose GPU of the Kepler generation with 2688 single precision cores. Only selected algorithms have been implemented for this platform, with a focus on tasks related to scientific visualization. The third target platform is Intel’s Xeon Phi coprocessor with 61 physical and 244 logical cores. This hybrid accelerator platform closes the circle as it represents the result of convergence of classical multi-core CPUs and GPU-based accelerators. Despite
its young history, it has gained much renown thanks to its combination of flexibility, efficiency, and performance. The least time has been spent in porting algorithms to this platform, because a) it was only recently released and b) the main intention behind its appearance in this thesis is proof of concept, i.e., its purpose is to demonstrate the compatibility of the algorithms and data structures with this newest generation of accelerators.

I close the many-core section with a brief description of the target platforms, in which I mostly highlight their differences and similarities. I mention some characteristic numbers, but since the chosen platforms serve as mere representatives of hardware classes, I will not indulge in a discussion of technical detail. Although the theoretical floating point peak performance will be given, the problems of interest are typically memory bandwidth bounded and FLOP rates are therefore the wrong metric here. Finally, since developing for the Kepler platform requires a different programming model, the discussion of the second target platform is a bit more detailed and includes a brief introduction to OpenCL.

**Platform I: Intel Sandy Bridge-EP**

Released in 2012, Intel Sandy Bridge-EP (SNB-EP) is the Intel Xeon-based server platform for the Sandy Bridge microarchitecture. The following compilation of relevant technical details was taken from the official platform data sheet[^75] as well as various sources related to the product websites of the Intel Xeon E5 processor family[^1]. SNB-EP generally features 2–4 sockets with 2–8 cores each, but the focus is here only on the target configuration, the dual socket system with two eight-core Intel Xeon E5-2680 server CPUs running at 2.7 GHz (see sketch in Fig. 2.2). Each physical core contains two logical cores supporting Intel Hyper-Threading, Intel’s own variant of SMT. Sandy Bridge was the first platform to support the Advanced Vector Extensions (AVX), an instruction set defining operations on either four double precision (DP) or eight single precision (SP) floating point values stored in 256 bit wide vector registers. The target system achieves a theoretical DP floating point peak performance of around 330 GFLOPS. Each core is equipped with 16 AVX registers, 32 KB on-chip L1 data cache, and 256 KB on-chip L2 data cache. SNB-EP implements a fully coherent cache protocol, combining each socket’s 20 MB L3 cache to a total of 40 MB last level cache. Two QPI channels offer an inter-socket communication rate of 16 GT/s. Four memory channels per socket give a maximum theoretical bandwidth of 102.4 GB/s for eight installed 1600 MHz DDR3 RAM modules. In [^66], 74 GB/s were actually measured for this platform via a manually tuned version of the STREAM benchmark [^87].

[^75]: http://ark.intel.com/products/family/59138/Intel-Xeon-Processor-E5-Family

The code developed for this platform is exclusively written in C++ (C++11 standard). The compute kernels rely on AVX intrinsics for explicit vectorization. OpenMP pragmas are used to introduce thread level parallelism. Intel MPI is used for communication on
Figure 2.2.: The SNB-EP system under consideration is a dual CPU system with eight 2.7 GHz cores per socket. A fully coherent cache protocol is implemented between both NUMA nodes with around 40 MB of combined last level cache. The theoretical peak memory bandwidth is at around 102.4 GB/s when 1600 MHz DDR3 RAM is used on all eight available memory channels.

the node level. All programs have been compiled with the Intel Compiler 14.0.1, following recommendations from miscellaneous optimization and compiler guides provided by Intel (e.g., [74]).

Platform II: Nvidia GPU Accelerators

The Nvidia K20x is the general purpose graphics processing unit (GPGPU) among the three target platforms and requires a bit more explanation, also because its use requires a different programming model. The K20x belongs to Nvidias newest generation of devices code-named Kepler GK110. Information about the architecture was to most parts extracted from the architectural whitepaper [102]. As is sketched in Fig. 2.3, the K20x comes with 14 activated Streaming Multiprocessors (SMX) which can be considered NUMA sockets with extended functionality. An SMX mounts 192 in-order SP cores clocked at 732 MHz. The architecture is still optimized for computer graphics, and so every group of three SP cores shares one DP floating point unit (DPU), resulting in a DP performance (theoretical peak at 1.25 TFLOPS) that is roughly three times lower than SP performance. The structure of the SMXs is the reason why GPUs are extremely efficient SMT devices. Each SMX can accommodate up to 2048 active threads, which are scheduled on the 192 physical cores in turns, aiming at latency hiding for any occurring memory access. Full occupancy of the SMXs is generally desirable when aiming for optimal performance of the GPU and is achieved by avoiding SMX-local resource conflicts between active threads. An SMX is equipped with 64K SP registers and 48 KB scratch pad memory, resources which are shared by all threads. Although the actual distribution of these resources is quite flexible and can be influenced programmatically,
on average this leaves every thread with at most 32 SP registers and 24 byte of scratch pad memory. Exceeding these limits results in a lower number of active threads and swapping between on-chip memory and main memory. In addition, each SMX has a private 16 KB L1 cache and 48 KB fast read-only data cache. A fully coherent 1.5 MB L2 last level cache is shared by all 14 SMXs. The platform supports 6 GB GDDR5 main memory, which is connected via 6 memory channels offering a combined theoretical peak bandwidth of 250 GB/s. Again, the actual value measured in [66] via the SHOC benchmark [32] is much lower at 168 GB/s.

Classical programming languages are neither cut out for the programming of GPGPUs nor are these languages directly supported. The K20x is an Nvidia CUDA device whose full potential can be released via the CUDA API [100], but its standard features are also exposed through OpenCL [91, 103], which is sufficient here. Besides, OpenCL has other benefits, as it relies on just-in-time (JIT) compilation: GPU code can be generated or modified at runtime. A simple but effective example are loop limits. They may only be known at program runtime, but still before kernel launch. Setting these limits to constant values in the kernel source code allows the JIT compiler to apply optimizations such as loop unrolling. Other, more general optimization techniques apply for OpenCL equally as for CUDA, and so I refer readers to Nvidia’s ample documentation regarding performance guidelines and best practices for the CUDA platform (e.g., see [100, 103, 99, 101, 98]). A short summary of the OpenCL programming model follows now.
OpenCL

OpenCL [91][103] is closely related to Nvidia’s CUDA [100] and adopts its *Single Instruction Multiple Threads (SIMT)* paradigm to describe data and task parallelism. OpenCL kernels (functions called on the GPU) strongly resemble classical computer graphics shaders, i.e., the program formulation is a sequential one and typically revolves around a single data primitive being processed. In shaders, these primitives are vertices and pixels. In OpenCL kernels, they can be anything, for instance elements of a vector. The sequential SIMT kernel formulation is parameterized with a linear index (a thread ID), that uniquely identifies each kernel instance within a group of concurrently launched kernels. A kernel instance corresponds to a GPU thread, and by means of the index it can identify its dedicated share of the work. Such a kernel instance is called *work item* in OpenCL terminology; the group of work items it belongs to is called *work group*. It is easy to see how a group of four work items identified by indices 0–3 can add vectors of length four in a SIMD fashion, when each work item takes care of one vector element. Thread level parallelism exists between work groups. A second similar work group of four work items could simultaneously compute something entirely different, the cross product of the vectors, perhaps. This logical structure can be perfectly mapped to the hardware sketched in Fig. 2.3. Up to 2048 active work items organized in work groups reside on an SMX and are scheduled on the 192 SP cores in turns, sharing the on-chip memory and register resources among them. The simplicity and efficiency of the hardware design imposes some additional constraints for programmers. The following list completes this short discussion of the K20x platform:

• One instruction unit serves 32 SP cores, i.e., 32 work items form a *warp* and are simultaneously stepped. The warp size denotes the width of the vector units and therefore defines the minimum work group size (32 on Nvidia GPUs).

• Divergent conditional branches are possible for the work items of a warp, but they result in serialized execution of the single branches and should be avoided.

• The SIMT approach may implicitly trigger expensive *gather* operations when data is loaded from arbitrary memory locations into the threads’ registers. Ensuring coalesced memory access is the key to minimizing the number of memory transactions and thus reaching peak performance. This rule also holds for writing data back to memory, as the mirroring *scatter* operation can turn out even more costly than *gather*.

• As indicated in Fig. 2.3, the 64 KB of fast on-chip memory are reconfigurable. CUDA allows to switch between three modes for the attribution of L1 cache vs. scratch pad memory: 16–48, 32–32, and 48–16. Unfortunately, these specifics are not reflected in the OpenCL API, fixing the configuration to 16 KB L1 cache and 48 KB scratch pad (which is in fact perfect for the kernels presented in later chapters).
If an SMX’s limits of scratch pad are exceeded, the data is transparently swapped to main memory causing severe performance penalties. If on the other hand work items exceed their full occupancy limit of 32 registers, the additional registers will not be swapped. Instead the number of active work items will be decreased.

The OpenCL standard offers primitives to synchronize the work items of a work group. Synchronizing work groups is so far only possible from the host (CPU) side, a device (GPU) side mechanism is planned for a future version of the standard.

Platform III: Intel Xeon Phi

The third target platform is the Intel Xeon Phi coprocessor. It represents the latest step in the development of mainstream accelerator platforms and can be considered the result of convergence of general purpose CPUs and specialized accelerators (e.g., GPUs). As such, the Xeon Phi is used to prove that the presented software solutions are portable, also and in particular to platforms that seem to define future hardware trends.

The model used in the tests is the Intel Xeon Phi 7120p, which was released in mid 2013. A brief summary captures the most relevant architectural features as listed by the official platform data sheet [76]. The Intel Xeon Phi 7120p has 61 physical in-order cores based on a design used in the first Pentium architecture. The cores run at 1.238 GHz (in Turbo up to 1.333 GHz) and each core features four contexts for SMT, accumulating to a total of 244 logical cores. Integrated in the cores, double width vector processing units (VPU) supporting fused multiply-add operate on 512 bit wide vector registers (8 DP or 16 SP floating point values). The VPUs support the full AVX2 instruction set, which means that in contrast to the SNB-EP’s VPUs they support *gather* and *scatter* instructions, making the Xeon Phi a complete SIMD architecture. Thanks to the wide vector registers the platform’s DP peak performance amounts to 1.2 TFLOPS. Concerning data storage, each core has 32 AVX2 registers, 32 KB L1 and 512 KB L2 cache, both on chip. As is indicated in Fig. 2.4 a bidirectional on-chip interconnect, the so-called *Core Ring Interface (CRI)*, combines all L2 caches to a fully coherent 30.5 MB last level cache shared by all cores. Like all memory requests, cache misses first travel the CRI until one of eight attached memory controllers processes them. Each memory controller is connected via two channels to the GDDR5 main memory, which results in a combined theoretical peak bandwidth of 352 GB/s. Again, actual measurements are found in [66], where the same benchmark tools are used as for the other target platforms. The tested device, a Xeon Phi 5110p, is a predecessor of the 7120p and has a theoretical peak bandwidth of 320 GB/s. Both devices use the same number of memory controllers, which is why the measured value of 165 GB/s (ECC enabled) can be expected to scale linearly to a value of around 180 GB/s for the 7120p.

For the programming of the Intel Xeon Phi platform basically the same tools are employed as for the SNB-EP architecture: SIMD parallelism is exploited by means of
intrinsics (a specialized subset of AVX2 this time) and thread level parallelism by means of OpenMP pragmas. The implementations ported to the Xeon Phi do not make use of node level parallelism. The Intel Compiler 14.0.1 is used for compilation, and the program optimization process has been mainly guided by specific platform documentation available from the Intel website (e.g., [77]).

2.2. A Pragmatic Introduction to Sparse Grids

Sparse grids – under that name – were first used in 1991 by Zenger [132] for the solution of the Poisson Equation. The theory behind, however, reaches back to 1963, when similar approximation schemes were introduced for interpolation and quadrature by Smolyak [121]. Since then, sparse grids have come a long way and have been employed in a variety of settings, among which there are the mentioned classical interpolation and quadrature tasks [49, 80, 45], data mining [48, 47, 107], and the solution of complex partial differential equations such as the Black-Scholes Equation for option pricing [112, 118]. Newer applications are found in uncertainty quantification [42, 96]. The grids’ distinguishing feature is to this day unique: The sparse grid technique is the only grid-based general purpose discretization technique applicable in higher dimensions.

The underlying idea of the cheap sparse grid discretization can be abstracted via the following example: A continuous line interval in 1-D can be discretized inexpensively using either one sampling point in the middle or two points at both ends. Lifting both discretization schemes to \( d \) dimensions reveals the difference in cost, as \( 1^d \ll 2^d \) for
large $d$. The exponential growth depending on $d$ observed for the two-point variant was coined *curse of dimensionality* by Bellman in 1961 [10]. Sparse grids stick closer to the one-point variant, taking advantage of a suitable set of hierarchical basis functions associated with the grid points. Sparse grids discretize the $d$-dimensional unit hypercube level-wise, starting with a single grid point on level 1 at the hypercube’s center. The accuracy of discretization can be flexibly controlled by adding points level-wise at the dyadic positions, following a scheme that ensures an optimal cost-benefit ratio [20]. Figure 2.5 uses two- and three-dimensional sparse grids of different levels to give a visual clue regarding this process.

The efficient hierarchical approach used in sparse grids weakens the dependency of the grid’s size on the dimensionality considerably. Instead of $N^d$ points, as are spent for a full grid with isotropic resolution $N$, the corresponding sparse grid merely spends $O\left(N \cdot \log^{d-1} N\right)$ points. The approach thus leads to manageable numbers of grid points also for more than three dimensions and delays the negative effects of the curse of dimensionality. Of course, these benefits come at some cost. Although the matter of accuracy would usually involve the discussion of Ansatz functions, it is still obvious that in the general case a sparse grid must lose accuracy over a full grid when it discards the majority of grid points. The effect can become drastic, especially in the approximation

---

In Bellman’s work, the term describes the exponential effort $O\left(\varepsilon^{-\alpha d}\right)$ one has to spend (e.g., in form of discretization points), in order to ensure a given accuracy $\varepsilon$ in $d$ dimensions ($\alpha > 0$ is problem dependent, influenced for instance by the smoothness of the treated function). Today, the term is used more freely, generally referring to the exponential dependency between a $d$-dimensional full grid’s size and the number $d$ of dimensions.
of non-smooth functions, or mathematically speaking, functions whose mixed second
derivatives are unbounded. This does, however, not render sparse grids useless for all
non-smooth real world scenarios, as the problem only exists for the grids’ pure, regular
form. Adaptive sparse grids can be derived from the regular form very naturally, ridding
the technique of its limitation.

The emphasis of this work is on sparse grid algorithms and data structures. For this
reason, I believe a thorough discussion of the grids’ structural properties serves readers
better in the understanding of this work than an extensive recapitulation of the mathe-
matical theory. Consequently, I begin with a purely structural analysis of the point sets
under consideration while readers are still unbiased by any notation or mathematical
formalism. In the analysis, I explain how different perspectives on the grids relate to
each other. Subsequently, all constructs are formalized and the notation is established
that is used throughout the remainder of this thesis. This process involves touching the
underlying mathematical theory and the discussion of Ansatz functions. Finally, I give
an overview of various structural variants of sparse grids.

2.2.1. Point Set Construction

The construction of sparse grids, as indicated in Fig. 2.5 can be approached from sev-
eral directions. In the following, sparse grids are first considered as multi-dimensional
binary-tree-like structures. The tree perspective directly reflects the grids’ hierarchical
construction, and it is best suited to uncover the grids’ inherent recursiveness and self-
similarity. From the trees, I will bridge towards another perspective, where a regular
sparse grid is seen as a superposition of full grids. Although it lacks support for fully
adaptive grids, the full grid perspective is important and forms the basis of the majority
of data structures presented in later chapters. In the examples, I focus primarily on
the 2-D case, as it favors visualization and, once understood, can be easily extended to
arbitrary dimensions.

Figure 2.6 uses a sketch of a 2-D sparse grid of level 3 to illustrate the recursive embed-
ding of smaller grids. The example is picked up again in the leftmost image of Fig. 2.7,
only now a tree representation is used, in which tree nodes represent grid points, and tree
levels correspond to sparse grid levels (starting with level 1 for the root at the top). Note
the typical sparse grid pattern arising from the vertical projection of the tree nodes onto
the base plane. The color coding is the same in both images: The \( d \)-dimensional subtrees
of level \((n − 1)\) are blue, the \((d − 1)\)-dimensional subtree of level \(n\) on the main axis is
green. From left to right in Fig. 2.7, the recursive construction is employed to step-wise
increase the tree (grid) level by one. Note that beside the drawn red links many other
parent-child relationships with respect to the horizontal (left to right) dimension exist,
but are left out to better highlight the recursive construction. For further studies of the
recursive composition, I recommend the array of grids in Fig. 2.5. Last but not least, the
recursive formulation provides a tool to express any complex higher-dimensional sparse
2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

Figure 2.6.: A $d$-D sparse grid of level 1 consists of one point. A $d$-D sparse grid of level $n > 1$ is composed of two $d$-D sparse grids of level $(n - 1)$ (blue), and one $(d - 1)$-D sparse grid of level $n$ (green).

A 2-D sparse grid of level 3 is shown. As indicated in the blue subgrids, the self-similarity allows to apply the recursive construction in any direction.

Figure 2.7.: The image shows the recursive dependence between 2-D sparse grids of levels 3, 4, and 5. Green subtrees are of lower dimensionality and discretize the hyperplane bisecting the domain. Blue subtrees are embeddings of lower-level grids (cf. respective left neighbor image). They are inserted on level 2 via the red connectors, ensuring the same leaf level in all subtrees.

grid as collection of lower-dimensional (even one-dimensional) structures. This property is exploited very successfully in several implementations as detailed in the analysis of data structures in Sect. 3.3.

In the full grid perspective, a $d$-dimensional sparse grid is seen as a combination of anisotropic regular grids (see Fig. 2.8). Grids with the same number of points belong to the same sparse grid level and are found on the scheme’s diagonals. Close inspection of any two neighboring grids on the left reveals their hierarchical dependency: One grid has twice as many grid points and arises from the other through a step of binary refinement of all points along one of the dimensions. During refinement, the cells around grid points are split in the middle, and new grid points (children) are inserted at the centers of all subcells. The similarity to the recursive construction is highlighted through the use of consistent colors: The green filled shapes mark grid points on the main axis (1-D expansion), and the blue empty shapes mark the grid points of the lower-level
Figure 2.8.: The sparse grid discretization on the right can also be seen as the result of combination of the anisotropic regular grids shown on the left.

2-D subgrids in both parts of the bisected domain. Squares mark the roots of all three subgrids.

Embedded regular grids vs. interconnected trees – the discussion will stretch over this whole thesis. As will become clear, it is not important to decide which representation generally represents the better choice. It is important to remember their differences and similarities to decide which one is the better choice for a particular task.

### 2.2.2. Sparse Grid Theory and Notation

The following overview of sparse grid theory provides readers with all information needed to follow the remainder of this work. This introduction to the matter will explain things at the example of function approximation, without going as far as showing proofs. Readers interested in proofs are referred to [132, 18, 20]. For further reading about sparse grids I also recommend [46, 107]. The notation used in the explanations follows [20, 50].

The starting point for sparse grids is a one-dimensional multi-level basis, from which \(d\)-dimensional basis functions are derived as the tensor products of the one-dimensional functions. The resulting hierarchical tensor product expansion is then truncated according to a cost-benefit analysis. These steps are explained in the following at the example of the piecewise linear hat function basis and homogeneous boundary conditions. Both are simplifying assumptions made here to keep unnecessary complexity out of this introduction. Subsequent sections will show, however, the assumptions can easily be given up again.
2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

Figure 2.9.: The relation between function spaces $V_l$ of piecewise linear functions in 1-D and their subspaces $W_l$, $l \in \{1, 2, 3\}$ is graphically explained.

**Multi-Level Subspace Splitting in 1-D**

Consider the unit interval $\Omega$

\[ \Omega := (0; 1), \quad \tilde{\Omega} := \Omega \cup \partial \Omega, \quad (2.1) \]

with levels of dyadic regular grids $\Omega_l$ given as

\[ \Omega_l := \{(l, i) \mid l \in \mathbb{N}, 1 \leq i \leq 2^l - 1\}. \quad (2.2) \]

The constant mesh size in $\Omega_l$ is $h_l := 2^{-l}$, and the grid points $(l, i)$ have abscissas

\[ x_{l,i} := i \cdot h = i/2^l, \quad 1 \leq i \leq 2^l - 1. \quad (2.3) \]

Let $\phi : \mathbb{R} \to \mathbb{R}$ be the *mother function of all hat functions* given by

\[ \phi(x) := \begin{cases} 1 - |x| & \text{if } -1 < x < 1 \\ 0 & \text{else}, \end{cases} \quad (2.4) \]

\[ \phi_{l,i}(x) := \phi(2^l x - i), \quad (2.5) \]

with basis functions $\phi_{l,i}$ derived from it via translation and dilation as shown in (2.5).

Let further be $\Psi_l$ the common *nodal point basis* on level $l$ with degrees of freedom on $\Omega_l$ given as

\[ \Psi_l := \{ \phi_{l,i} \mid l \in \mathbb{N}, 1 \leq i \leq 2^l - 1\}, \quad \text{with} \quad (2.6) \]

\[ V_l := \text{span} \{ \Psi_l \} \quad (2.7) \]
being the function space it spans. \( V_l \) can be hierarchically decomposed into hierarchical increments \( W_{l'} \), \( l' \leq l \) given by

\[
W_{l'} := \text{span} \{ \phi_{l',i} \mid i \in I_{l'} \},
\]

(2.8)

\[
I_{l'} := \bigoplus_{i \mid i \in \mathbb{N} : 1 \leq i \leq 2^{l'} - 1, \text{i odd}} \prod_{l'' = 1}^{l' - 1} W_{l''}
\]

(2.9)

with index sets \( I_{l'} \) identifying those \( \phi_{l',i} \) that form the levels of a hierarchical basis. The construction ensures that functions in \( W_l \) vanish at all grid points \( x_{l',i} \) for lower levels \( l' < l \) as illustrated in Fig. 2.9. This leads to the following relation between spaces \( V_l \) and subspaces \( W_l \):

\[
V_1 = W_1
\]

\[
V_l = V_{l-1} \oplus W_l = V_1 \oplus W_2 \oplus \cdots \oplus W_l = \bigoplus_{l'=1}^{l} W_{l'}
\]

(2.10)

With both bases at hand, a function \( u \in V_n \) can now be uniquely expressed either with respect to the nodal point basis as

\[
u(x) = \sum_{i=1}^{2^n-1} u_i \cdot \phi_{n,i}(x), \quad u_i = u(x_{n,i}),
\]

(2.11)

or with respect to the hierarchical basis as

\[
u(x) = \sum_{l=1}^{n} \sum_{i \in I_l} \alpha_{l,i} \cdot \phi_{l,i}(x).
\]

(2.12)

The \( \alpha_{l,i} \in \mathbb{R} \) in (2.12) are generally denoted by hierarchical coefficients or (hierarchical) surpluses. The hierarchical transformation used to determine the \( \alpha_{l,i} \) is generally referred to as hierarchization, and it is the topic of a detailed discussion in Sect. 3.2.1 and Sect. 4.3.1.

Note that this whole construction process and its outcome bear a strong resemblance to what is called multi-resolution analysis in the context of wavelets \[33, 123, 9, 85, 86, 1\]. From the wavelet perspective, (2.4) can be seen as both, the mother wavelet and father wavelet, and with the scaling function encoded in (2.5) dual bases for the spaces \( V_l \) and \( W_l \) can be constructed. From the primary design criteria of wavelet bases only orthogonality between the spaces \( W_l \) is unfulfilled (although it is fulfilled with respect to the energy scalar product). It would be a desirable property for the hierarchical basis here, too, but it is subordinated to a simple function representation and interpolation capabilities. Ensuring orthogonality for wavelet bases often results in extremely complex basis functions without explicit form, rendering them unusable for interpolation tasks.

### Multi-Level Subspace Splitting in \( d \)-D

The generalization to \( d \) dimensions leads to the redefinition of \( \Omega \) as the \( d \)-dimensional unit hypercube given by

\[
\Omega := (0; 1)^d, \quad \bar{\Omega} := \Omega \cup \partial \Omega.
\]

(2.13)
In 2-D, the tensor products of the 1-D hat functions are bilinear pagoda functions, as seen here at the example of the basis functions $\phi_{(2,1),(1,1)}$ and $\phi_{(1,2),(3,1)}$ spanning the subspace $W_{(2,1)}$.

In an attempt to consistently adopt the 1-D notation for the $d$-D case, the scalar indices are exchanged for multi-variate indices. To this end, the following operations and norms for multi-variate indices are defined, as they help to avoid explicit references to the dimensionality and thus increase general readability:

\[ \vec{l} \leq \vec{i} \iff l_j' \leq l_j \text{ for all } j \in D \]
\[ \vec{\alpha} := (\alpha, \ldots, \alpha) \in \mathbb{R}^d \]
\[ 2^\vec{l} := (2^{l_0}, \ldots, 2^{l_{d-1}}) \in \mathbb{N}^d \]
\[ |\vec{l}|_1 := \sum_{j \in D} l_j \]
\[ |\vec{l}|_\infty := \max_{j \in D} l_j \]

with the set $D$ of indices in $d$ dimensions given by

\[ D := \{0, \ldots, d-1\} \]

Next, a $d$-dimensional multi-level basis is constructed from the tensor products of the one-dimensional hierarchical basis functions. The grid points, formerly identified by level-index pairs $(l, i)$, become level-index vector pairs

\[ (\vec{l}, \vec{i}) := ((l_0, \ldots, l_{d-1}), (i_0, \ldots, i_{d-1})) \quad \vec{l} \in \mathbb{N}^d, \vec{i} \in \mathcal{I}_\vec{l} \]

again defined by the index set $\mathcal{I}_\vec{l}$

\[ \mathcal{I}_\vec{l} := \{ \vec{i} \mid i_j \in \mathcal{I}_{l_j} \text{ for all } j \in D \} \]

The piecewise linear basis functions $\phi_{l,i}$ are multiplicatively combined to piecewise $d$-linear basis functions $\phi_{\vec{l},\vec{i}}$ (see the plot in Fig. 2.10 for an example in 2-D)

\[ \phi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{j \in D} \phi_{l_j,i_j}(x_j), \]
2.2. A Pragmatic Introduction to Sparse Grids

Figure 2.11.: The relation between function spaces $V_{\vec{l}}$ and $W_{\vec{l}}$ in 2-D is graphically explained via the so-called subspace tableau. The 1-D basis functions $\phi_{l,i}$ are sketched along the axes around the $V_{n}$ and $W_{l}$ hinting at the tensor products $\phi_{l,i}$ that form the 2-D basis. Inside each $W_{l}$, points mark the degrees of freedom of the $\phi_{l,i}$ and lines mark their disjoint supports.

spanning analogous spaces $V_{\vec{l}}$ and subspaces $W_{\vec{l}}$

\[
V_{\vec{l}} := \text{span} \left\{ \phi_{\vec{l},\vec{i}} \mid \vec{i} \leq \vec{l} < 2^{\vec{l}} \right\}, \tag{2.19}
\]
\[
W_{\vec{l}} := \text{span} \left\{ \phi_{\vec{l},\vec{i}} \mid \vec{i} \in I_{\vec{l}} \right\}. \tag{2.20}
\]

The relation between $d$-dimensional spaces $V_{\vec{l}}$ and subspaces $W_{\vec{l}}$ can be formulated as

\[
V_{\vec{l}} = \bigoplus_{\vec{i} \leq \vec{l}} W_{\vec{i}}. \tag{2.21}
\]

An illustration of relation (2.21) in two dimensions is provided in Fig. 2.11. Note therein the dimension independent notation $V_{n}$ that specifically refers to the space $V_{n} = V(n,...,n)$. Furthermore, Fig. 2.11 resolves where the full grid view on sparse grids shown in Fig. 2.8 of the previous Sect. 2.2.1 comes from. And finally, rewriting (2.12) for interpolation in $V_{n}$ via the $d$-dimensional hierarchical basis yields

\[
u(\vec{x}) = \sum_{\vec{i} \leq \vec{n}} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.22}\]
Approximation on Sparse Grids

The potential of the so far seen hierarchical data representation unfolds with an additional assumption of smoothness. Let $H^{\text{mix}}_2(\bar{\Omega})$ be the Sobolev space of functions over $\bar{\Omega}$ with bounded mixed second derivatives given by

$$H^{\text{mix}}_2(\bar{\Omega}) := \{ u : \bar{\Omega} \to \mathbb{R} \mid \nabla^{\bar{\alpha}} u \in L^2(\Omega), |\bar{\alpha}|_1 \leq 2, u|_{\partial\Omega} = 0 \}, \quad \text{with} \quad (2.23)$$

$$D^{\bar{\alpha}} u := \frac{\partial^{|\bar{\alpha}|} u}{\partial^{\alpha_0} x_0 \ldots \partial^{\alpha_{d-1}} x_{d-1}}. \quad (2.24)$$

Let the level $n$ of a subspace be defined as

$$n := |\bar{l}|_1 - d + 1, \quad (2.25)$$

and let its cost be measured as the number of associated degrees of freedom, leading to

$$|W_{\bar{l}}| = 2^{n-1} = 2^{|\bar{l}|_1 - d} = O(2^{|\bar{l}|_1}). \quad (2.26)$$

The classical sparse grid analysis shows for functions $u \in H^{\text{mix}}_2(\bar{\Omega})$, that the surplus $\alpha_{\bar{l},\bar{i}}$ is subject to an exponential decay with respect to the subspace level [20], more precisely

$$|\alpha_{\bar{l},\bar{i}}| = O(2^{-2|\bar{l}|_1}). \quad (2.27)$$

The analysis identifies the sparse grid space $V_n^{(1)}$ as cost-benefit-optimal regarding grid points spent and approximation accuracy achieved. The space is defined as

$$V_n^{(1)} := \bigoplus_{\bar{l} \in L_n^{(1)}} W_{\bar{l}}, \quad \text{with the level vector set} \quad (2.28)$$

$$L_n^{(1)} := \{ \bar{l} \in \mathbb{N}^d \mid |\bar{l}|_1 \leq n + d - 1 \}. \quad (2.29)$$

Graphically, (2.28) describes the truncation of the subspace tableau at the $n$-th (multi-dimensional) diagonal, which drops costly subspaces with $L_1$-norm $|\bar{l}|_1 > n + d - 1$ (thus the 1 in the notation’s superscript). Several impressions of this truncation have already been shown. In Fig. 2.11, for instance, the truncated subspaces $W_{\bar{l}} \subset V_3 \setminus V_3^{(1)}$ are grayed out, and a likeness of the resulting 2-D sparse grid of level 3 is shown in the lower left corner. In Sect. 2.2.1 on the other hand, the same grid structure arises from an incremental process of binary refinement of the leaf subspaces.

Concerning the accuracy of approximation of functions $u \in H^{\text{mix}}_2(\bar{\Omega})$, a sparse grid interpolant $\hat{u}_n \in V_n^{(1)}$ achieves an error (with respect to general $L_p$-norms) of

$$\|u - \hat{u}_n\|_p = O\left(h_n^2 \cdot n^{d-1}\right), \quad (2.30)$$

whereas a full grid interpolant $u_n \in V_n$ achieves only a slightly better result with

$$\|u - u_n\|_p = O\left(h_n^2\right). \quad (2.31)$$
2.2. A Pragmatic Introduction to Sparse Grids

The additional logarithmic factor in the error term is the price one has to pay for truncating the original approximation space. It is, however, a price worth paying for a considerably lowered number of grid points. An asymptotic comparison of numbers is given by

$$|V_n| = (2^n - 1)^d \quad \text{and} \quad |V_n^{(1)}| = \sum_{l'=0}^{n-1} 2^{l'} \cdot \left( d - 1 + l' \right) \quad = O \left( 2^{nd} \right),$$

while an impression of actual numbers is shown in Table 2.3. The table also lists the number of grid points spent for the classical extension of $V_n^{(1)}$ to the space $\bar{V}_n^{(1)}$ with explicit boundary representation on so-called trapezoidal boundaries (cf. Sect. 2.2.3). Compared to $V_n^{(1)}$, $\bar{V}_n^{(1)}$ yields a considerable increase in the number of points, but it still uses far less points than the full grid space $V_n$. To summarize, the sparse grid discretization mitigates the negative effects of the curse of dimensionality considerably, while almost preserving the accuracy of approximation for smooth functions $u \in H_2^{\text{mix}}(\Omega)$.

<table>
<thead>
<tr>
<th>$d$</th>
<th>$n = 5$</th>
<th>$n = 8$</th>
<th>$n = 5$</th>
<th>$n = 8$</th>
<th>$n = 5$</th>
<th>$n = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>129</td>
<td>1,793</td>
<td>257</td>
<td>2,817</td>
<td>961</td>
<td>65,025</td>
</tr>
<tr>
<td>3</td>
<td>351</td>
<td>7,423</td>
<td>1,505</td>
<td>21,249</td>
<td>29,791</td>
<td>16,581,375</td>
</tr>
<tr>
<td>4</td>
<td>769</td>
<td>23,297</td>
<td>7,681</td>
<td>133,889</td>
<td>923,521</td>
<td>4,228,250,625</td>
</tr>
<tr>
<td>5</td>
<td>1,471</td>
<td>61,183</td>
<td>36,033</td>
<td>754,945</td>
<td>28,629,151</td>
<td>$&gt;1.0 \times 10^{12}$</td>
</tr>
<tr>
<td>6</td>
<td>2,561</td>
<td>141,569</td>
<td>159,489</td>
<td>3,940,609</td>
<td>887,503,681</td>
<td>$&gt;2.7 \times 10^{14}$</td>
</tr>
<tr>
<td>7</td>
<td>4,159</td>
<td>297,727</td>
<td>676,161</td>
<td>19,418,369</td>
<td>$&gt;2.7 \times 10^{10}$</td>
<td>$&gt;7.0 \times 10^{21}$</td>
</tr>
<tr>
<td>8</td>
<td>6,401</td>
<td>580,865</td>
<td>2,772,225</td>
<td>91,479,297</td>
<td>$&gt;8.5 \times 10^{11}$</td>
<td>$&gt;1.7 \times 10^{24}$</td>
</tr>
<tr>
<td>9</td>
<td>9,439</td>
<td>1,066,495</td>
<td>11,064,033</td>
<td>415,563,777</td>
<td>$&gt;2.6 \times 10^{13}$</td>
<td>$&gt;4.5 \times 10^{21}$</td>
</tr>
<tr>
<td>10</td>
<td>13,441</td>
<td>1,862,145</td>
<td>43,186,689</td>
<td>1,831,708,161</td>
<td>$&gt;8.1 \times 10^{14}$</td>
<td>$&gt;1.1 \times 10^{24}$</td>
</tr>
</tbody>
</table>

Table 2.3.: The table compares the numbers of grid points spent for sparse grid discretization without and with (trapezoidal) boundaries, as well as full grid discretization without points on the boundary. Numbers are given for 2–10 dimensions and discretization levels $n = 5$ and $n = 8$. 

2.2.3. Hierarchical Bases

In the previous section, it is explained how a basis for multiple dimensions can be derived from one-dimensional piecewise linear hat functions. But tensor product bases are much more powerful than that. This section contains a brief overview of other commonly used bases that also play a role in later chapters. Special features are highlighted and
On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

illustrations are shown, but for information about approximation accuracy readers are referred to the literature cited.

Boundary Treatment for the Linear Hat Function Basis

One way to extend the classical formulation of the linear hat function basis for inhomogeneous boundary conditions is to introduce two boundary grid points on level 0 at positions

\[ x_{0,0} := 0, \quad x_{0,1} := 1. \]  

(2.34)

The corresponding Ansatz functions

\[
\phi_{0,0}(x) := \begin{cases} 
1 - x & \text{if } 0 \leq x < 1 \\
0 & \text{else,}
\end{cases}
\]

\[
\phi_{0,1}(x) := \begin{cases} 
x & \text{if } 0 < x \leq 1 \\
0 & \text{else}
\end{cases}
\]  

(2.35)

are linear in \( \bar{\Omega} \) and extend the hat function basis naturally. Function plots of \( \phi_{0,0} \) and \( \phi_{0,1} \) are shown in Fig. 2.12a. They appear again in the plots of the polynomial basis and the prewavelet basis (cf. Fig. 2.12c and Fig. 2.12d), for which they form the optional level 0 as well.

Explicit boundary treatment is straightforward in 1-D, but in \( d \)-D it again raises the obvious complexity issue: \( 2^d \) grows exponentially in \( d \). Two variants of sparse grid boundary treatment are explained graphically via the subspace tableau in Fig. 2.13. Trapezoidal or projected boundaries (center) are much less costly and are therefore generally preferred over diagonal boundaries (right), which consistently extend the notion of the diagonal to the boundaries of the subspace tableau (left). Both variants are generally found to behave similar regarding approximation quality, although this has to my knowledge not in fact been the topic of a published study. Moreover, the huge impact of boundary points on storage cost raises thoughts in the sparse grid community whether even further reduction of the boundary resolution might still lead to satisfying results. After all, Fig. 2.13 suggests and Tab. 2.3 confirms that – especially in higher dimensions – explicit boundary representation consumes the large majority of all grid points.

It is impossible to completely outrun the curse of dimensionality, but sometimes strategies present themselves that help to circumvent its immediate effects. Reformulating general boundary problems to problems with homogeneous boundary conditions is one strategy often practiced for Dirichlet boundaries in PDE settings. Another possibility is to embed the problem under consideration into a larger domain and use “far” zero-boundaries. Yet another method relies on implicit boundary representation, extrapolating interior function values towards the boundary (cf. classification in Sect. 5.1). The prerequisite to the latter approach is an adjusted set of basis functions, for example
2.2. A Pragmatic Introduction to Sparse Grids

(a) The linear hat function basis allows for explicit boundary representation on level 0.

(b) The modified linear hat function basis extrapolates linearly towards the boundaries.

(c) On level $n$, the higher order polynomial basis uses polynomial Ansatz functions of degree $p = n + 1$.

(d) The prewavelet basis is constructed to ensure $L^2$-orthogonality for Ansatz functions on different levels.

Figure 2.12.: The image shows four different hierarchical bases in 1-D which are commonly used with sparse grids.
The subspace tableau on the left also includes the subspaces associated with the boundary functions $\phi_{0,0}$ and $\phi_{0,1}$. The sparse grid spaces $V_3^{(1)}$ for trapezoidal boundaries (green dashed line, center image) and $\tilde{V}_3^{(1)}$ for diagonal boundaries (blue dashed line, right hand side image) are sketched.

Excellent results have been achieved by means of the modified linear hat function basis, especially in settings related to data mining [107, 69]. Note that other bases (for instance the polynomial basis) have been successfully altered in a similar way (e.g., in [107]), too, but they are less frequently used and not discussed here.

**The Polynomial Basis of Higher Order**

The piecewise polynomial basis functions are introduced in [18] as a generalization of the piecewise linear hat functions. They follow the concept of hierarchical Lagrangian interpolation, i.e., each function $\phi_{l,i}^{(p)}$ is a polynomial of degree $p$, with roots at the degrees of freedom of its $p$ direct hierarchical ancestors, and with function value 1 at the abscissa $x_{l,i}$ of its own associated degree of freedom. The polynomial degree is denoted by $p$, and on level $l$ it is usually given by $p = l + 1$. A fixed value $p_{\text{max}}$ can, however, be chosen, such that $p = \min(l + 1, p_{\text{max}})$ holds. The hierarchy thus consistently starts with (optional)
linear functions on level 0 and a quadratic parabola on level 1. Let

$$\text{anc}((l, i), k) := (l - k, 2 \cdot \left\lfloor \frac{i}{2^{k+1}} \right\rfloor + 1), \quad 0 \leq k < l,$$

be the ancestor of grid point \((l, i)\) on level \(l - k\). The 1-D Ansatz functions are then defined as

$$\phi_{l,i}^{(p)}(x) := \left\{ \begin{array}{ll}
\eta_{l,i} \cdot x(x - 1) \prod_{k=1}^{p-2} \left( x_{\text{anc}((l,i),k)} - x \right) & \text{if } \frac{i-1}{2^l} \leq x \leq \frac{i+1}{2^l} \\
\tilde{\phi}_{l,i}^{(p)}(x) & \text{else,}
\end{array} \right.$$ (2.38)

$$\eta_{l,i} := 1/\tilde{\phi}_{l,i}^{(p)}(x_{l,i}).$$ (2.39)

Normalization of the polynomials at their degrees of freedom is ensured via (2.39). As usual, \(d\)-D basis functions are obtained as tensor products of 1-D functions, only now the vector \(\vec{p} \in \mathbb{N}^d\) is additionally used for control of the polynomial degree in each dimension:

$$\phi_{l,i}^{(p)}(\vec{x}) := \prod_{j \in D} \phi_{l,i_j}^{(p_j)}(x_j).$$ (2.40)

The 1-D basis is plotted in Fig. 2.12c. The polynomial \(\phi_{l,i}^{(p)}\) and the linear hat \(\phi_{l,i}\) have identical supports and their degrees of freedom sit in the same spot, however, \(\phi_{l,i}^{(p)}\) is continuously differentiable everywhere within its support. The polynomial functions of higher order are designed for accurate approximation of particularly smooth functions, as the formulated smoothness assumption is even stricter than the one that applies for the hat functions. Moreover, the functions’ design prevents oscillations even despite possibly unlimited polynomial degrees, and so none of the \(\phi_{l,i}^{(p)}\) exceeds a function value of 1.117 \((1.117^d \text{ in } d\)-D) within its support [18] (cf. horizontal line indicated in Fig. 2.12c).

The Prewavelet Basis

The dyadic prewavelet basis was specifically designed for the solution of higher-dimensional elliptic PDEs via tensor product bases. It is based on [30] and was first employed on sparse grids in [58]. Following the idea of wavelets, the basis functions are constructed with the postulation of a compact function support and \(L_2\)-orthogonality between different levels. Furthermore, a prewavelet on level \(l\) is demanded to be a composition of a minimum number of neighboring hat functions \(\phi_{l,i}\) from the generating system on level \(l\).

For better discrimination between linear hat functions \(\phi_{l,i}\) and prewavelets, let the latter for now be denoted by \(\psi_{l,i}\). The simplest linear combination of hat functions satisfying
2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

the above requirements is given by

\[
\begin{align*}
\psi_{0,0} &:= \phi_{0,0}, \\
\psi_{0,1} &:= \phi_{0,1}, \\
\psi_{1,1} &:= \phi_{1,1},
\end{align*}
\tag{2.41a}
\]

\[
\psi_{l,i} := \begin{cases} \\
\frac{9}{10} \phi_{l,1} - \frac{6}{10} \phi_{l,2} + \frac{1}{10} \phi_{l,3} & \text{for } l > 1, i = 1 \\
\frac{9}{10} \phi_{l,2l-1} - \frac{6}{10} \phi_{l,2l-2} + \frac{1}{10} \phi_{l,2l-3} & \text{for } l > 1, i = 2^l - 1 \\
\frac{1}{10} \phi_{l,i-2} - \frac{6}{10} \phi_{l,i-1} + \phi_{l,i} - \frac{6}{10} \phi_{l,i+1} + \frac{1}{10} \phi_{l,i+2} & \text{else (for } l > 1\).
\end{cases}
\tag{2.41b}
\]

The basis functions for levels 0 and 1 are the same as for the hat function basis, as stated by (2.41a) and displayed in the top image of Fig. 2.12d. On each higher level, the basis functions split into a set of symmetric interior functions (empty on level 2, cf. center image of Fig. 2.12d), and two asymmetric near-boundary functions associated with the degrees of freedom closest to the boundary on either end of the domain (2.41b). Since the basis functions are constructed as prewavelets (not as wavelets), orthogonality is ensured between the levels,

\[
V_n = \bigoplus_{l=1}^n W_l \quad W_j \perp W_k \text{ for } j \neq k,
\tag{2.42}
\]

which is paid by the price of overlapping non-orthogonal functions of larger supports within each level, such that

\[
\psi_{l,i} \not \perp \psi_{l,j} \quad \text{for } \psi_{l,i}, \psi_{l,j} \in W_l : |i - j| \leq 4.
\tag{2.43}
\]

Note also that the boundary extension lacks full support for the orthogonality feature, as we have

\[
\begin{align*}
\psi_{l,1} &\not \perp \psi_{0,0} \quad \text{for } l \geq 1, \\
\psi_{l,2l-1} &\not \perp \psi_{0,1} \quad \text{for } l \geq 1.
\end{align*}
\tag{2.44}
\]

As a consequence of the postulates stated for its construction, the prewavelet basis spans the same space as the linear hat function basis for almost all kinds of sparse grids (only for spatially adaptive grids this is not necessarily true). The basis is rather unsuited for interpolation settings, as the common hat functions lead to the same approximation and have a less sophisticated structure and smaller supports. The prewavelets’ only true calling is the solution or preconditioning of PDEs (e.g., see [58, 95, 37, 38]), where the orthogonality between levels leads to sparse mass and stiffness matrices. As shown in later chapters, the orthogonality property also facilitates modifications in certain sparse grid algorithms, sometimes leading to a tremendous reduction of program runtime.

### 2.2.4. Structural Variants

I conclude the introduction to sparse grids with an overview of the different structural manifestations of this discretization technique. According to structural and algorithmic
2.2. A Pragmatic Introduction to Sparse Grids

resemblance, I group them into three major categories. The first two, *dimensionally adaptive sparse grids (DASG)* and *spatially adaptive sparse grids (SASG)*, directly reflect the intrinsic hierarchical nature of the grids, both in algorithms and data structures. One can speak of a direct sparse grid approach. The third one, the combination technique (CT), only uses the sparse grid approximation space for the extrapolation of problem solutions of higher accuracy. The CT avoids a hierarchical representation of data, and so one can speak of an indirect approach to sparse grids.

**Dimensionally Adaptive Sparse Grids (DASG)**

The term DASG stands for all sparse grid variants whose point set is defined by a set of level vectors. Sparse grid theory is based on the grids’ classical form, the *regular sparse grids*, for which the only constraint is the global sparse grid level. Their level vector set \( \mathcal{L}_n^{(1)} \) is already introduced in (2.29).

In literature, several references to *truncated sparse grids* and *anisotropic sparse grids* are found, often denoting very similar things. In case of anisotropic sparse grids [47] for instance, the usual hyperplane-cut in the subspace tableau at \(|\vec{l}|_1 = n + d - 1\) can be customized by means of an additional vector defining the tilt of the cutting plane. References to truncated sparse grids appear in [93, 25, 92].

The level vector set of regular sparse grids \( \mathcal{L}_n^{(1)} \) is therein further constrained by a truncation vector \( \vec{c} \in \mathbb{N}^d \), which is added to the notation’s subscript next to the global level constraint \( n \). Component \( c_j, j \in \mathcal{D} \) of the truncation vector specifies the level at which the refinement tree is cut off in dimension \( j \) (cf. item (iv) in Fig. 2.14). Such grids are used in Sect. 5.2 to hierarchically decompose full grids. The level vector set \( \mathcal{L}_{n, \vec{c}}^{(1)} \) of truncated sparse grids is given by

\[
\mathcal{L}_{n, \vec{c}}^{(1)} := \mathcal{L}_n^{(1)} \setminus \left\{ \vec{i} \in \mathbb{N}_0^d \mid \vec{i} \not\leq \vec{c} \right\}.
\]

The superclass of subspace-based sparse grids are the *dimensionally adaptive sparse grids* themselves (sometimes also called *dimension-adaptive sparse grids*) [64, 49, 45]. The only constraint for the level vector set of a DASG is rather a convention. In order to ensure proper hierarchical linking of all subspaces – which is convenient in many sparse grid algorithms –, all grid points’ hierarchical ancestors must be in the grid, too. To this end, \( \mathcal{L} \) (2.47) describes a set of “suprema”, which mark those leaf subspaces in the grid that are not refined in any of the dimensions. The corresponding level vector set \( \mathcal{L}_{\vec{c}} \) is given by

\[
\mathcal{L}_{\vec{c}} := \left\{ \vec{i} \in \mathbb{N}_0^d \mid \exists \vec{i}_{\text{max}} \in \mathcal{L} : \vec{i} \leq \vec{i}_{\text{max}} \right\}, \quad \text{with} \quad (2.46)
\]

\[
\mathcal{L}_{\vec{c}} \subset \left\{ \vec{i} \in \mathbb{N}_0^d \right\}.
\]

\(^8\)In the context of the CT, there is another reference to *truncated sparse grids* in [11, 117]. Here, a truncation vector \( \vec{t} \) disallows combi-grids \( \vec{i} \not\in \vec{t} \). The only meaningful way to translate this condition to the DASG context might be to exclude leaf subspaces falling below the imposed thresholds.
Figure 2.14.: Several kinds of sparse grids are put into perspective in the scheme of spaces and subspaces. (i) The relation of full grid spaces and hierarchical increments $V_{1,3} = W_{1,1} \oplus W_{1,2} \oplus W_{1,3}$ is indicated with yellow shading in the top row. (ii) A comparison is shown for a DASG (left) and CT grid (right) defined by the same set $\bar{L} = \{(4, 1), (3, 2), (2, 3)\}$ (cf. (2.47)). It can be seen that the approximation spaces are equal if the same type of Ansatz functions is used in both grids. (iii) On the right, each combi-space $V_{\vec{l}}$ is annotated with its CT weight inside a circle. The red $V_{\vec{l}}$ bear negative weights to balance out the redundancy introduced by overlaps of the blue $V_{\vec{l}}$. (iv) The blueish dashed lines suggest how a truncation vector $\vec{c} = (4, 3)$ could limit the refinement per dimension, a constraint specifiable for DASG and CT grids alike. (v) The DASG is extended to an SASG through introduction of the grid points $p_1 = (4, 3), (9, 3)$ and $p_2 = (4, 3), (3, 7)$. Both points are marked as red crosses, as are their parents $((3, 3), (5, 3)$ and $(4, 2), (9, 1)$ for $p_1$, $(3, 3), (1, 7)$ and $(4, 2), (3, 3)$ for $p_2$), which are needed to ensure a grid point hierarchy without gaps.
Note finally how $\hat{L}$ (2.47) is the only remaining constraint for the level vector set of a DASG $L_L$ (2.46), which is why all other constraints are dropped from the notation (cf. (2.29) and (2.45) for regular and truncated grids, respectively). An illustration of a DASG with $\hat{L} = \{(4,1), (3,2), (2,3)\}$ is shown in Fig. 2.14 (cf. (ii) therein).

**Spatially Adaptive Sparse Grids (SASG)**

One of the major benefits of sparse grids is their inherent support for adaptivity. Adaptivity can help to overcome the smoothness assumptions of regular sparse grids while keeping the number of grid points relatively low. The tree perspective presented in Sect. 2.2.1 gives intuitive access to the adaptive side of the grids. Each tree node is the root of a self-similar subtree, which makes local adaptive refinement seem very natural. In addition to the algorithmic and structural straightforwardness, sparse grids offer a built-in adaptivity criterion in form of the hierarchical surplus. Furthermore, for many non-hierarchical grid types the price of adaptivity either consists in ensuring grid conformity or it consists in a sophisticated handling of hanging nodes. Sparse grids, in contrast, directly operate on a multi-level hierarchy without having to satisfy any topological constraints.

SASG are the most flexible sparse grid type. Isotropic and anisotropic refinement of single grid points, thinning out via coarsening, dynamic switching between different refinement criteria – there are no limits to the creativity with which one can design refinement operators. Similar to DASG, the only guideline is to ensure a hierarchy without gaps. Following the example of the condition for DASG in (2.46) and (2.47), $\mathcal{P}_P$ specifies a set of leaf grid points that are not refined in any dimension. The grid $\mathcal{P}_P$ then contains the points in $\bar{P}$ as well as all their hierarchical ancestors and is given by

$$\mathcal{P}_P := \left\{ (\vec{l}, \vec{i}) \in N^d_0 \times \mathcal{I}_I \mid \exists (\vec{l}_{\text{max}}, \vec{i}_{\text{max}}) \in \mathcal{P} : \vec{i} \leq \vec{l}_{\text{max}} \land \vec{i} \leq \vec{i}_{\text{max}} \right\}; \quad \text{with} \quad (2.48)$$

$$\bar{P} \subset \left\{ (\vec{l}, \vec{i}) \in N^d_0 \times \mathcal{I}_I \right\}. \quad (2.49)$$

Figure 2.14 shows how the introduction of single grid points $p_1$ and $p_2$ turns a DASG into a more general SASG. In order to satisfy (2.48), the missing parents of both points need to be inserted into the grid, too (cf. item (v) in Fig. 2.14).

But all this praise for SASG must be put into perspective. After 25 years of effort spent on the development of efficient software for SASG, these grids currently face their greatest challenge: In addition to common challenges for adaptive numerical codes, such as load balancing in irregular structures, SASG codes must also handle global data dependencies introduced by the hierarchical approach. This is especially tough, as these dependencies often cause random memory access in traditional complexity-optimal algorithms and inhibit parallelism almost completely on any level. These circumstances take their toll on modern computer hardware and call urgently for a new set of algorithms and data structures. I will mention and evaluate a few recent efforts in the examination of data structures and algorithms in Sect. 3.3 and Sect. 4.3.2.
The Combination Technique (CT)

An early appearance of the idea behind the CT is already found in Smolyak’s 1963 publication [121]. Still, the discovery of the combination technique (especially under that name) is credited to Griebel and Zenger for their ground-breaking publication [59] in 1992. The CT represents a totally different approach to sparse grids. Let \( \mathcal{L} \) be a valid level vector set according to (2.46). A sparse grid can be seen as combination of hierarchical increments \( W_{\vec{l}}, \vec{l} \in \mathcal{L} \) (given in form of full grids), but analogously it can be seen as a superposition of coarse full grids \( V_{\vec{l}}, \vec{l} \in \mathcal{L} \), too. These full grids are generally referred to as component grids or simply combi-grids. The idea behind the CT is that, regardless of the underlying problem, several coarse grid solutions of lower accuracy can be combined to produce a global solution of higher precision. To this end, solutions to a problem are first computed on all coarse combi-grids. These solutions are then reduced via weighted summation, wherein a suitable choice of weights resolves the redundancy issue introduced by points present in more than one combi-grid. The reduction often produces an extrapolated solution of comparable accuracy as obtained when directly solving in the combined approximation space of all combi-grids – which is essentially the approximation space of a DASG defined by the same level vector set \( \mathcal{L} \). The principle is visualized at the example of a 2-D sparse grid in Fig. 2.14 (cf. items (i)–(iii)). The CT’s robustness and quality of solution are further increased with the introduction of the opticom (“optimal combination technique”) [65, 44] around 2005. The opticom presents a strategy to determine an optimal, problem-dependent set of reduction coefficients that extends the CT’s applicability and performance.

The CT yields extremely attractive properties for today’s HPC community: The combi-grid solutions only interact during the reduction phase and can therefore be computed in an embarrassingly parallel fashion. Above all, the CT does without making strict assumptions regarding the origin of the combi-solutions. Legacy codes optimized for full grids can therefore be recycled in this step. For all these reasons, the CT has gained much popularity since its invention and has been successfully employed in a broad range of settings [59, 127, 17, 112, 80, 81]. From this set, I want to point out scientific visualization [127]. In contrast to classical sparse grids, treatment of the CT’s full combi-grids has always been favored by graphics hardware. However, with the hierarchical transformation presented in Sect. 5.2, I present a hierarchical alternative. To my knowledge, it is also the first successful attempt of visualizing direct sparse grids. Finally, the CT is currently considered the only way to lift sparse grids to the exa-scale. Consequently, a focus of current research related to the CT is failsafe computing in the presence of fault-prone hardware on large clusters. A question often tackled is how to recover from the case of a missing combi-solution [82, 63, 108].

The aspect considered the main disadvantage of the CT is its lacking support for spatial adaptivity, but flexible handling of dimensional adaptivity suffices to compensate for this flaw in most cases. Furthermore, the method still diverges in some settings (e.g., regression, cf. [43]), and so users still often push the limits of the technique (and their
luck) in a trial-and-error fashion.

2.3. Sparse Grid Software – An Overview of Publicly Available Tools and Libraries

The third and last part of this chapter is formed by an overview of sparse grid software currently circling the scientific community. The list of software contains implementations that are published on the web for free download and use. All implementations are assessed with respect to functionality provided and fitness for the modern age of parallel compute platforms. The search has come to the unfortunate but inevitable conclusion, that even in the year 2014 sparse grids themselves are still a sparse subset among the publicly available grid-based codes. The search results are organized such that implementations are listed under the same item, if they offer similar functionality, or if they are or were developed by the same group. Note that although some implementational aspects are already mentioned in this section, the detailed discussion of data structures is postponed until Sect. 3.3.

2.3.1. SG++

The SG++ toolbox[10] is probably the most powerful set of sparse grid functions currently available. Its emergence in the context of SASG is described in [107], but the code base has continuously been extended ever since its first release in late 2006. SG++ excels in many aspects. Developers are supplied with a vast collection of materials concerning installation, quick start, class documentation, and code snippets, all available for download under a BSD-style license. Its feature line-up is also unsurpassed, due to a large community of contributing users and developers from a broad range of fields. The back end code of SG++ is written in C++, but bindings for Python, Java, and MATLAB exist.

The authors of SG++ have spent much effort in a modular code design with room for customized extensions. The code is structured into several modules, keeping the core functionality separated from advanced features and application specific components. At the heart of the implementation, an unordered map is employed for storing the structure of the SASG. However, the design allows for the substitution of the map with other data structures, which is in fact planned for an upcoming release in late 2014. High flexibility is ensured through the use of operations, which can be defined for different grid types and tasks. Frequently used operations are for example hierarchical transformations, grid

---

[9] This limitation to freely accessible codes is born out of necessity, but to my knowledge the discussion includes all high performance sparse grid codes currently appearing in publications.

2. On Sparse Grids in a Heterogeneous Landscape of Computing Platforms

evaluation, and the computation of differential operators, but customized refinement operators can also be added to this list. Finally, various ansatz functions are supported in SG++ as further detailed in [107].

With its major contributors being from the sector of scientific computing, SG++ is developed with the ambition to reach maximum performance. All algorithms have been thoroughly examined regarding parallel performance on multiple levels. The hash map as central component, however, has revealed itself as one of the main performance blockers, and its current form is reconsidered as underlying data structure. Alternative formulations have meanwhile been explored successfully, and although not directly listed as part of the common SG++ release, highly tuned cluster codes on the basis of simplified data structures originate from the SG++ code base (cf. discussions in Sect. 3.3, Sect. 5.1 and Sect. 5.3).

The SG++ project has taken major influence on the sparse grid code developed in the context of this thesis. The modular structure of the SG++ code has been mirrored, with the long term goal in mind to eventually migrate the new algorithms and data structures into the toolbox. The integration is currently planned for the SG++ release in late 2014.

2.3.2. Tasmanian, PECOS & Co.

Several libraries and toolkits related to sparse grid quadrature and interpolation are developed at Sandia (SNL) and Oak Ridge National Labs (ORNL). The website of SG-MGA [11] (Sparse Grid Mixed Growth Anisotropic Rules) lists a collection of numerical toolkits (e.g., SANDIA_SGMGG, SANDIA_SPARSE, SMOLPACK, SANDIA_RULES, all developed at SNL and downloadable under the GNU Lesser General Public License), related to the construction of isotropic and spatially adaptive sparse grids, and numerical quadrature using higher-order polynomials on Chebyshev points. The theory behind is based on [23, 121, 9, 97, 96], and the development was further inspired by the functionality of the SPINTERP library (cf. Sect. 2.3.4).

PECOS [12] (Parallel Environment for Creation Of Stochastics) is a C++ library developed at SNL within a larger project, namely the DAKOTA project [13] (Design Analysis Kit for Optimization and Terascale Applications), which aims at answering questions concerning parameter sensitivities, uncertainties, optimization, and calibration for computational models given as input. PECOS makes internal use of SGMGA & Co. and offers self-adapting numerical integration capabilities for multi-dimensional approximating functions. To this end, hierarchical and CT versions of dimensionally adaptive sparse grids are implemented in small integration driver classes, with specialized functionality limited to quadrature. After free registration, the PECOS code can be downloaded and

11 http://people.sc.fsu.edu/~jburkardt/m_src/sgmga/sgmga.html
13 http://dakota.sandia.gov
2.3. Sparse Grid Software – An Overview of Publicly Available Tools and Libraries

distributed under the GNU Lesser General Public License.

Tasmanian (Toolkit for Adaptive Stochastic Modeling and Non-Intrusive Approximation) is a C++ class-based library developed at ORNL, with a focus on high-dimensional integration and interpolation based on adaptive higher-order sparse grid discretization schemes with non-equidistant (mostly Chebyshev) points. The source code and documentation are freely distributed under the GNU General Public License on the project website. According to the SGMGA website, part of the functions developed in SGMGA & Co. have also been integrated into Tasmanian. In addition to the functionality regarding Chebyshev polynomials provided by the SNL libraries, Tasmanian offers basic support for wavelets as introduced in [55, 61].

All three libraries, SGMGA, PECOS and Tasmanian, have a high-level toolkit character and aim at usability, offering bindings for C, C++, Fortran 90 and MATLAB. The SGMGA API is a C-style interface giving access to a collection of sophisticated numerical subroutines, but not the underlying data structures and algorithms. Tasmanian’s API is modeled in a class hierarchy, but the level of control exposed to the user is also limited to the selection of interface functions, as no obvious extension points are defined. In PECOS, extending the class hierarchy seems more natural, but the code’s sole purpose, numerical quadrature, clearly defines the design of all classes.

Summing up, it can be said that all these libraries clearly focus on functionality rather than performance. The adaptive sparse grid data structures revolve around simple point collections rather than sophisticated multi-dimensional constructs. Consequently, most algorithms involving grid evaluation or integration iterate the point sets in a straightforward manner. Thread level parallelism via OpenMP pragmas is included in Tasmanian, but as annotations in the source code indicate, ample opportunity for algorithmic optimization is still given. Although the “P” in PECOS suggests otherwise, the library’s sparse grid classes rely on strictly sequential code.

2.3.3. fastsg and sgbench

fastsg and sgbench are sparse grid codes freely distributed under the GNU General Public License.15 sgbench is a set of benchmark routines for testing the performance of data structures and basic algorithms optimized for regular sparse grids as proposed in [91].

The fastsg project is the result of the continued research on low-level sparse grid optimization in heterogeneous environments originally started with sgbench. Although not directly linked on the website, all project code has also been ported to GPGPUs and is accessible through the author. fastsg wraps highly optimized implementations of basic

\[ \text{http://tasmanian.ornl.gov/index.html} \]
\[ \text{http://www.lrr.in.tum.de/~murarasu/} \]
sparse grid tasks (e.g., hierarchical transformations on and evaluation of sparse grids) into a small C++ library. At the core of the library is a SparseGrid class for regular and truncated sparse grids as detailed in [93]. The functionality has been extended with auto-tuning capabilities for load balancing in distributed heterogeneous environments as described in [92].

fastsg is a slim library with the ambition to exploit modern hardware features to the fullest. ILP, TLP, and NLP are all present in the library, however, not all of them are considered in every algorithm. fastsg does not offer a vast selection of features, but it yields a clear, simple design and can thus be seen as a starting point for optimized codes for non-adaptive sparse grids.

2.3.4. SPINTERP

SPINTERP [80] is a MATLAB library that is widely known and used in the sparse grid community. The code was published online together with extensive documentation [79] under a customized, liberal license already in 2004, and it was maintained by the author until 2008. The original publication [80] is also archived in the ACM TOMS Collected Algorithms as Algorithm 847. SPINTERP is still popular, which can be attributed to its good documentation, a clear and slim code design, and features frequently demanded in sparse grid codes. On top of this, it is an efficient, easy-to-use software, that lets the large community of MATLAB users conveniently explore and exploit some of the advanced capabilities of sparse grids.

The implementation is based on the combination technique, giving the user access to basic functionality such as interpolation and quadrature, but more sophisticated features such as self-regulating dimensional adaptivity and searching extrema for optimization problems are in the portfolio as well. Besides the classical multi-linear grids on dyadic discretization points, the user can also choose non-equidistant point sets for use with Chebyshev polynomials of higher order.

Despite the consistent use of MATLAB’s “lower-level” programming interface (e.g., function `feval`) and the formulation of efficient algorithms, the topic of parallelism is entirely skipped in the source code as well as the publications. The mentioning of “vectorization” in the documentation of several functions can be misleading, as it denotes in MATLAB jargon the avoidance of loops by indexing into vectors and tensors using index ranges. Under the hood, these techniques allow the MATLAB interpreter various optimizations, e.g., precompiled library code can take care of costly computations.

It can for instance be downloaded from http://www.netlib.org/toms/index.html

16
3. Co-Design, Part I: Formulating the Challenges

What makes developing data structures and algorithms a real challenge is the mutual dependence of the two, and this insight is exactly the crux of this work: I design data structures and algorithms in a tightly coupled process, in which I try to adapt them to each other in the best possible way while maintaining clear interfaces that do not hinder performance. The goal of this approach is to prevent the development process from becoming a one-way street, in which the algorithm itself becomes the specification for the data structure, or where a new algorithm is fully subjected to the capabilities of an existing data structure. Such circumstances tend to result in highly specific, non-reusable data structures or algorithms with poor performance.

One of the major difficulties in the process of co-design is that there is no handbook that explains how to carry it out in a structured way. The interdependence of all components rules out a full dissection, and what is more, this equally holds for the preceding problem analysis and the comprehensible and coherent description of the steps afterwards. Chapters 3 and 4 do therefore not only serve to present results of this thesis; they can also be seen as guideline through all layers of the complex design process. On the top level, the discussion is divided into two main parts:

1.) The contents of Chapt. 3 are purely analytic. Looking for efficient solutions, I systematically approach the problem under consideration from three different directions. First, I analyze sparse grid theory with respect to general constraints it imposes on sparse grid implementations. I then concentrate on the actual tasks to be mastered, and, with a close look at data access patterns, I discuss the options regarding these tasks’ efficient completion. Finally, I conduct an extensive study of data structures available for all kinds of sparse grids and extract these data structures’ most desirable properties.

2.) In Chapt. 4 I evaluate and combine the conclusions from the analytic chapter in order to derive solutions. I present data structures and algorithms and put them to test in extensive performance experiments. Since usability is a guiding concern in the design process, I also look at the code structure and report on experiences made with it so far.

Before entering the analysis, I want to point out that this chapter represents a substan-
tial contribution by itself. I feel this is necessary, because while the significance of a newly discovered numerical method or algorithm reveals itself to readers almost automatically, this process does not necessarily take place for an analysis such as the one conducted here. The added value of this chapter consists in a systematic compilation and preparation of facts and ideas carefully extracted from sparse grid theory, algorithms, and data structures. I want to specifically emphasize that all revisited algorithms and formulas are presented in a clear and original form that should enable readers to directly translate the given rules into working programs. In addition, this chapter contains the most comprehensive comparison of sparse grid data structures so far published. One of this study’s major contributions is the first-time formalization of some of the examined constructs.

3.1. Algorithmic Implications of the Sparse Grid Construction

This short section opens the analysis of sparse grid algorithms and data structures with a glance back at the condensed sparse grid theory of the previous chapter. With the aim of efficient implementations in mind, I give a brief summary of observations concerning the sparse grid construction. The purpose of this section is to formulate questions and to draw general conclusions from the underlying theory rather than to analyze a particular sparse grid task or form.

3.1.1. Multi-Recursive Tree Structure

Sparse grids are infamous for their multi-dimensional, multi-recursive tree structure. It is intuitively clear from the illustrations of parent-child relationships in Fig. 2.7 that honoring the interwoven structure of dependencies in algorithms is a difficult task (especially considering that the illustration only shows an incomplete subset of dependencies). In fact, the implications of these recursive dependencies for sparse grid data structures are most critical and immediately raise urgent questions, some of which are:

- How is this multitude of dependencies best represented with acceptable overhead?
- What will be the cost and what will be the bottleneck of grid traversal?
- Does the multi-dimensional nature of the problem admit effective use of caches?
- Will descriptions be feasible for simple bandwidth devices (accelerators), too?

I will proceed with more implications of the sparse grid construction, leaving these important questions unanswered for now for readers to contemplate on.
3.1. Algorithmic Implications of the Sparse Grid Construction

3.1.2. A Benefit of Tensor Products: One-Dimensional Schemes

Consider the probably simplest problem that occurs in sparse grid settings: determining the integral of a sparse grid interpolant \( u_h \). Let this \( u_h \) be given by the consistent set of grid points \( P \) (cf. (2.48)) and coefficients \( \alpha_{\ell,i} \). The integral of \( u_h \) is defined by

\[
U_{\Omega}^h := \int_{\Omega} u_h \, d\Omega = \int_{x_{d-1}=0}^{1} \cdots \int_{x_0=0}^{1} u_h(x_0, \ldots, x_{d-1}) \, dx_0 \cdots dx_{d-1}
\]  

(3.1a)

\[
= \int_{x_{d-1}=0}^{1} \cdots \int_{x_0=0}^{1} \sum_{(\ell,i) \in P} \alpha_{\ell,i} \cdot \phi_{\ell,i}(x_0, \ldots, x_{d-1}) \, dx_0 \cdots dx_{d-1}
\]  

(3.1b)

\[
= \sum_{(\ell,i) \in P} \alpha_{\ell,i} \int_{x_{d-1}=0}^{1} \cdots \int_{x_0=0}^{1} \phi_{\ell,i}(x_0, \ldots, x_{d-1}) \, dx_0 \cdots dx_{d-1}.
\]  

(3.1c)

The limits of the integrals over the supports of the basis functions \( \phi_{\ell,i} \) (and also over the domain \( \Omega \)) are constant with respect to all variables \( x_j, j \in D \), which permits to flexibly swap all integrals in the formula. This is irrelevant when integrating simple hat functions \( \phi_{\ell,i} \), as their integrals are directly given by \( 2^{-|\ell|} \) which makes implementing formula (3.1c) straightforward. Now consider the more sophisticated example of the polynomial basis on some higher-dimensional transformed domain, for which precomputing each \( \phi_{\ell,i} \)'s integral is out of question. Because of the growing computational load per element it could pay off to devise a more elaborate scheme that sweeps over all basis functions in several passes, incrementally computing and applying 1-D integration operators efficiently in one dimension after the other. Such an algorithm would rather follow formula (3.1b), taking advantage of efficient 1-D schemes.

As mentioned before, integration is only the simplest example. The tensor product structure of the sparse grid bases permits treatment of much more complex problems following the same principle. On the one hand, Sect. 3.2.1 summarizes how the logic behind multi-dimensional transformations can be broken down to defining a set of one-dimensional dependencies between grid points in a 1-D operator. Just like integration, the full transformation then only requires the application of this operator in a loop over all problem dimensions. On the other hand, Sect. 3.2.3 revisits how even tasks as complex as the application of stiffness matrices in multiple dimensions can be accomplished via the so-called unidirectional principle, which “merely” requires the formulation of corresponding 1-D operators.

Only having to implement 1-D operators is, admittedly, a very attractive prospect bearing obvious advantages ranging from a more comprehensible problem description to better software maintainability. Moreover, the much dreaded dimensionality \( d \), with which the complexity of a problem and its solution process typically rises and falls, suddenly becomes a mere parameter to the solution algorithm. The next paragraphs will show, however, that there is a trade-off for these benefits.
3. Co-Design, Part I: Formulating the Challenges

3.1.3. Multi-Pass Processing and Changing Data Dependencies

So far, this sections has only highlighted the upsides of completing computational tasks in distinct passes associated with the single dimensions. Now, the negative aspects as well as their implications are addressed.

The first limitation is directly given by the separation of computational passes itself. It seems likely (and it is indeed true), that if schemes are strictly one-dimensional, mixing the application of 1-D operators in different dimensions will provoke inconsistency of the data. Regarding concurrency, this means that phases of safe simultaneous processing always end when moving on to the next pass, unless additional mechanisms exist to assure data consistency. The ambition of writing high performance code thus depends critically on two optimization criteria, namely

1) the computational efficiency of the 1-D schemes, and

2) the parallel efficiency of these schemes’ application.

Another consequence not easily dealt with is the repeated change of data dependencies. A famous (or better infamous) toy problem that serves to illustrate the dilemma is Gauß elimination for the direct solution of a system of \(N\) linear equations. Although the algorithmic complexity of \(O(N^3)\) operations is left unaffected, the swapping of the three nested loops tends to have a heavy impact on the solver’s performance. The effect can be attributed to the cache efficiency of the implementation. It is usually highest, if the fastest index moves with small strides in contiguous memory. These are bad news for programmers of sparse grid algorithms, as the underlying theory usually forces loops with respect to all dimensions. In other words, typical tasks require both the fastest and the slowest loop to be executed. Maximizing the degree of data locality is therefore imperative for a high performance implementation. But this is tricky again, since satisfying global data dependencies, as they commonly occur in hierarchical schemes, often entails scattered data access.

3.2. The Core Tasks

The previous Sect. 3.1 contains a recapitulation of facts about sparse grid construction. It is intended to raise readers’ awareness of general limitations (and opportunities) of sparse grid implementations, while readers are still unbiased by knowledge about the actual problems to solve. Now, the focus is on three common tasks on sparse grids that frequently reappear as subtasks in sparse-grid-based applications. The major goal of this section is to learn about data dependencies that often govern the actions and force certain data access patterns when solving these tasks.

First, I focus on hierarchical transformations, also often denoted by the name hierar-
3.2. The Core Tasks

chization. I discuss the theory behind three different 1-D operators, namely those for the transformations to the linear hat function basis, the polynomial basis, and the prewavelet basis. Second, I analyze the factors influencing the process of interpolation (also called evaluation) on a sparse grid. The algorithm for interpolation deviates somewhat from the usual scheme relying on 1-D operators, as completion of the task can be accomplished in a single pass. Third and last, the topic is the unidirectional principle, the underlying concept for more sophisticated operators on sparse grids, such as the application of a stiffness matrix to a vector. Same as for the transformations, the prerequisite here is the formulation of 1-D operators, which are applied to the grid in a certain order defined by the so-called UpDown scheme. A separate part at the end of this section summarizes all observations made regarding data access patterns in the treated algorithms.

3.2.1. Basis Transformations

Basis transformations are a fundamental part of nearly all sparse grid applications. As mentioned in Sect. 3.1, the transformations between the tensor product bases are defined as one-dimensional schemes, which are applied in a loop over all dimensions. At this point it is therefore possible to completely ignore $d-1$ dimensions of the problem and fully focus on the formulation of 1-D operators.

I begin the discussion with the most prominent and simple example, the change from the piecewise linear nodal point basis to the hierarchical hat function basis. I proceed with the generalized form of the piecewise linear basis, the piecewise polynomial basis of higher order. Both transformations turn sampled function values into hierarchical coefficients, the surpluses, which is why they are commonly referred to as hierarchization. This name seems unsuitable for the third object of study, the transformation to the prewavelet basis, as it takes a set of linear surpluses as input. Moreover, the inverses of the hierarchization schemes are trivial, which is why I can confidently skip their discussion. Turning prewavelet coefficients back to linear surpluses, on the other hand, is less straightforward.

Consider a set of input values $u_{l,i}, 0 \leq l \leq n \in \mathbb{N}_+, i \in \mathcal{I}_l$ and a corresponding set of hierarchical surpluses $\alpha_{l,i}$. The general relation between the input and output values of the transformations is given by the formula

$$\alpha_{l,i} = u_{l,i} - \sum_{(l',i') \neq (l,i)} \alpha_{l',i'} \cdot \phi_{l',i'}(x_{l,i}).$$

(3.2)

Obviously, this formula leads to a system of $N = 2^n + 1$ linear equations, which takes $\mathcal{O}(N^3)$ operations to solve. In the following, I explain how the task of computing the $\alpha_{l,i}$ can be accomplished in much shorter time for all examined bases.
3. Co-Design, Part I: Formulating the Challenges

Figure 3.1.: From left to right, the bottom up scheme implied by formula (3.7) is illustrated. Spaces and affiliations of coefficients are color coded: red stands for full grid approximations, blue for hierarchical increments. Surpluses $\alpha_{l,i}$ (solid blue) are computed in place by subtracting the approximation $v_{l-1}(x_{l,i})$ (dashed red) from function values $u_{l,i}$ (solid red). On level 0, we already have $\alpha_{0,i} = u_{0,i}$ as indicated on the right.

The Linear Hat Function Basis

The definition of the left hierarchical neighbor (3.3) as the closest ancestor of a grid point to its left adds a spatial component to the ancestor relationship defined in (2.37). Let the right hierarchical neighbor be defined analogously (3.4). For an illustration, go back to Fig. 2.9 and observe that $\text{left}(3,5) = (1,1)$ and $\text{right}(3,5) = (2,3)$.

$$\text{left}(l,i) := \begin{cases} 
(0,0) & \text{if } i = 1 \\
\text{anc}(l,i), \text{ffs}(i-1) - 1 & \text{else}, 
\end{cases} \quad (3.3)$$

$$\text{right}(l,i) := \begin{cases} 
(0,1) & \text{if } i = 2^l - 1 \\
\text{anc}(l,i), \text{ffs}(i+1) - 1 & \text{else}, 
\end{cases} \quad (3.4)$$

where $\text{ffs}$ denotes the common C function that returns the one-based index of the least significant bit set in the binary representation of a natural number $n$, i.e.,

$$\text{ffs}(n) := 1 + \arg\min_{b \in \mathbb{N}} \{ (2^b \text{ AND } n) \neq 0 \}. \quad (3.5)$$

Let $w_l = \sum_{i \in I_l} \alpha_{l,i} \phi_{l,i} \in W_l$, $l > 0$ be the increment improving the lower level approximation $v_{l-1} = \sum_{l' < l} w_{l'} \in V_{l-1}$ of function $u$ in between the interpolation points $x_{l',i}$, $l' < l$ of $v_{l-1}$. The $x_{l,i}$ on level $l$ are inserted at dyadic positions between the lower level $x_{l',i}$, each $x_{l,i}$ sitting exactly in the middle between its hierarchical neighbors. The surpluses $\alpha_{l,i}$ on level $l$ are defined as the function values $u_{l,i} = u(x_{l,i})$ minus the contribution of the piecewise linearly interpolated approximation $v_{l-1}(x_{l,i})$ of $u_{l,i}$, i.e.,

$$v_{l-1}(x_{l,i}) = \frac{v_{l-1}(x_{\text{left}(l,i)}) + v_{l-1}(x_{\text{right}(l,i)})}{2} = \frac{u_{\text{left}(l,i)} + u_{\text{right}(l,i)}}{2}, \quad \text{and thus (3.6)}$$

$$\alpha_{l,i} := u_{l,i} - v_{l-1}(x_{l,i}) = u_{l,i} - \frac{u_{\text{left}(l,i)} + u_{\text{right}(l,i)}}{2}. \quad (3.7)$$
Formula (3.7) specializes the general relation (3.2) for the hat function basis and can be applied to determine the surpluses in arbitrary order, unless an in-place algorithm is to be devised that operates directly on the input data. Then, a bottom up scheme as illustrated in Fig. 3.1 is implied, starting on level \( n \), working its way up to level 1, determining the \( \alpha_{l,i} \), \( l = n, \ldots, 1 \) along the way. The algorithmic steps are also explicitly listed in lines 2–6 of the following section’s Alg. 1.

### The Polynomial Basis of Higher Order

Being its generalization, the piecewise polynomial basis of higher order shares a few properties with the linear hat function basis. One of these properties is that basis functions have disjoint supports on each level of the hierarchy, leading to the same abstract formula for the surpluses as for the hat function hierarchy:

\[
\alpha_{l,i} := u_{l,i} - \alpha_{0,0} \cdot \phi_{0,0}(x_{l,i}) - \alpha_{0,1} \cdot \phi_{0,1}(x_{l,i}) - \sum_{k=1}^{l-1} \alpha_{\text{anc}(l,i),k} \cdot \phi_{\text{anc}(l,i),k}(x_{l,i})
\]  

(3.8)

For the hat functions, inexpensive linear interpolation could be used to come up with the more optimized formula (3.7). No such optimization offers itself for the polynomials, and all dependencies wired in (3.8) need to be resolved explicitly. With the successive scheme proposed in [18, 20], there is, however, an alternative to a direct implementation of (3.8). After applying an initial linear hierarchization step, the scheme localizes all operations (in a hierarchical sense), by only updating grid point coefficients using the coefficients of the points’ direct parents. I rewrite the central formula from [18] as (3.9) and derive the following efficient in-place scheme:

\[
\begin{align*}
\alpha^{(p)}_{l,i} & := \begin{cases} 
\alpha_{l,i} \text{ according to (3.7)} & \text{for } p = 1 \\
\alpha^{(p-1)}_{l,i} & \text{for } p > l \\
\alpha^{(p-1)}_{l,i} - \vartheta_{p,j} \cdot \alpha^{(p-1)}_{\text{anc}(l,i),1} & \text{else, with } j = i \mod 2^p,
\end{cases}
\end{align*}
\]  

(3.9)

where the \( p \) in the superscript of the \( \alpha^{(p)}_{l,i} \) denotes the order of the polynomial associated with the surplus. The \( \alpha^{(p)}_{l,i} \) are computed incrementally for increasing order \( p \), starting with linear surpluses \( \alpha^{(1)}_{l,i} = \alpha_{l,i} \) (cf. (3.7)), and ending with surpluses \( \alpha^{(l+1)}_{l,i} \) (or \( \alpha^{(p_{\text{max}})}_{l,i} \) for a given maximum polynomial degree \( p_{\text{max}} \leq l + 1 \)). The coefficients \( \vartheta_{l,i} \) remain unchanged during the generation of these sequences and can be precomputed for \( l \in \{2, \ldots, n\}, i \in \mathcal{I}_l \) via

\[
\begin{align*}
\vartheta_{l,i} & := \begin{cases} 
\frac{z - x_{l,i}}{1 - z} \cdot \frac{x_{l,i}}{z} \prod_{k=2}^{l-1} \frac{x_{l,i} - x_{\text{anc}(l,i),k}}{z - x_{\text{anc}(l,i),k}} & \text{for } 1 \leq i < 2^{l-1} \\
\vartheta_{l,2^{l-1} - i} & \text{else,}
\end{cases}
\end{align*}
\]  

(3.10a)

with \( z = x_{\text{anc}(l,i),1} \) being the abscissa of \( (l,i) \)’s direct parent. Translation of (3.9) into an actual algorithm results in another bottom up scheme as shown in Alg. 1. For clarification, three things should be mentioned:
3. Co-Design, Part I: Formulating the Challenges

- The symmetry in (3.10b) not only halves the number of stored $\vartheta_{l,i}$, exploiting it is actually essential here, as (3.10a) does not hold for $i > 2^{l-1}$.

- The modulo operation used in line 10 of Alg. 1 takes advantage of the discretization’s self-similarity introduced by the choice of dyadic points. The quotients of relative distances between grid points and their ancestors (as they occur in the original formula in [18] and are still recognizable in (3.10a)) are invariant with respect to the grid level. This enables the optimized scheme, in which the value of $p$ is fixed in the loop from line 8 of the algorithm, i.e., the same set of $\vartheta_{p,j}$ is used to update the surpluses on all grid levels.

- Surpluses of the highest degree $p = n + 1$ only occur on level $n$, but because $\alpha_{n+1}^{(n)}(\alpha_{n}^{(n)})$ (cf. (3.9)) holds, the loop from line 7 of Alg. 1 stops at polynomial degree $n$.

Algorithm 1

The (potential in-place) algorithm computes the surpluses $\alpha_{l,i}^{(p)}$ for the piecewise polynomial basis of higher order in a bottom up scheme. Code lines 2–6 compute the linear surpluses (order $p = 1$) of the input, and with each iteration of the loop in line 7, the order of the surpluses is increased by one.

1: function hierarchizePolynomial({$u_{l,i}$}; {$\vartheta_{l,i}$})
2: for $l = n, n - 1, \ldots, 1$ do ▷ compute linear surpluses
3: for $i = 1, 3, \ldots, 2^l - 1$ do
4: $\alpha_{l,i} \leftarrow u_{l,i} - 0.5(u_{l,i} + u_{l,i})$
5: end for
6: end for
7: for $p = 2, 3, \ldots, n$ do ▷ compute higher order surpluses
8: for $l = n, n - 1, \ldots, p$ do ▷ compute surpluses of order $p$
9: for $i = 1, 3, \ldots, 2^l - 1$ do
10: $j \leftarrow i \mod 2^p$
11: $\alpha_{l,i} \leftarrow \alpha_{l,i} - \vartheta_{p,j} \cdot \alpha_{\text{anc}((l,i),1)}$
12: end for
13: end for
14: end for
15: return {$\alpha_{l,i}$} ▷ return final surpluses
16: end function

The Prewavelet Basis

In 1-D, linear hierarchization is a prelude to both, the prewavelet transform and the higher-order polynomial hierarchization. In $d > 1$ dimensions, however, the prewavelet transform expects a preceding linear hierarchization with respect to all dimensions, whereas polynomial hierarchization simply takes $d$ applications of Alg. 1 as is. Similar to the filtering properties of other wavelet transforms, the prewavelet transform also
3.2. The Core Tasks

(a) Illustration of (3.12)

(b) Construction of sums from (3.11) and (3.13)

Figure 3.2.: Figure 3.2a demonstrates the splitting (3.12) of non-basis function $\phi_{l,k}$ (here: $\phi_{l+1,2i}$) into a linear combination of three basis functions. Figure 3.2b explains the construction (3.11) of temporary values $t_{l,k}$ (pink) and the formula (3.13) for determining the linear surpluses $\alpha_{l,i}$ (blue). Information is propagated bottom up through the $t_{l,k}$, which are further split and processed according to (3.12). This is indicated by the pink and blue tridents at the top, which use a coloring consistent with the illustration in Fig. 3.2a.

computes hierarchical averages and details. The result is a non-interpolating decomposition with mutual dependencies between overlapping prewavelet functions on each level. Although this means that the solution of an SLE is a necessary subtask, the next paragraphs show that the algorithmic complexity of the transform is still much lower than the complexity of solving the general transformation formula (3.2).

An understanding of the prewavelet transform is best developed by analyzing its inverse first. Let $\gamma_{l,i}, 1 \leq l \leq n, i \in I_l$ be the prewavelet coefficients to a corresponding set of linear surpluses $\alpha_{l,i}$. For each grid point $(l', i), l' < l$, a temporary variable $t_{l,k}, k = i \cdot 2^{l-l'}$ (“at” its projection on level $l$) recursively sums up the contribution of all prewavelets on levels $l' \geq l$ as defined in (3.11) (compare with the prewavelets’ definition (2.41b) and the arrows converging at the pink bullets in Fig. 3.2b).

$$t_{l,k} := \begin{cases} 0 & \text{for } l > n \\ -\frac{6}{10}(\gamma_{l,k-1} + \gamma_{l,k+1}) + t_{l+1,2k} & \text{else (3.12a)} \end{cases}$$

As implied by (2.41b), the $t_{l,k}$ denote weights of non-basis hat functions, which can be
3. Co-Design, Part I: Formulating the Challenges

represented as linear combinations of actual basis functions (3.12) (see also Fig. 3.2a):

\[ \phi_{l+1,2k} = \phi_{l,k} \]

\[ - \frac{1}{2} (\phi_{l+1,2k-1} + \phi_{l+1,2k+1}) \]

(3.12a)

(3.12b)

Counting the occurrences of basis function \( \phi_{l,i} \) in the hat function and the prewavelet representation establishes the relation between the two. For this, recall the prewavelets’ composition of hat functions (2.41b) and use the two-level relation (3.12) to rewrite non-basis hat functions \( \phi_{l,i} \pm 1 \). The dependencies lead to three equations (3.13) for near-boundary and interior basis functions \( \phi_{l,i} \) (for \( l > 1 \)), and they can also be retraced visually in Fig. 3.2b. Together, (3.11) and (3.13) define an efficient bottom up leapfrog scheme for the inverse transform, in which the \( t_{l,k} \) and \( \alpha_{l,i} \) are used to determine each other in alternating order.

\[ \alpha_{l,i} = \begin{cases} 
\frac{9}{10} \gamma_{l,1} + \frac{1}{10} \gamma_{l,3} + t_{l+1,2} - \frac{1}{2} t_{l,2} & \text{for } i = 1 \\
\frac{9}{10} \gamma_{l,2i-1} + \frac{1}{10} \gamma_{l,2i-3} + t_{l+1,2i(2i-1)} - \frac{1}{2} t_{l,2i-2} & \text{for } i = 2^{l-1} \\
\gamma_{l,i} + \frac{1}{10} (\gamma_{l,i-2} + \gamma_{l,i+2}) + t_{l+1,2i} - \frac{1}{2} (t_{l,i-1} + t_{l,i+1}) & \text{else} 
\end{cases} \]

(3.13)

\[ \begin{align*}
\frac{12}{10} \gamma_{l,1} + & \frac{4}{10} \gamma_{l,3} = \alpha_{l,1} - t_{l+1,2} + \frac{1}{2} t_{l+1,4} \\
\frac{4}{10} \gamma_{l,i-2} + & \frac{16}{10} \gamma_{l,i} + \frac{4}{10} \gamma_{l,i+2} = \alpha_{l,i} - t_{l+1,2i} + \frac{1}{2} t_{l+1,2(i+1)} \\
\frac{4}{10} \gamma_{l,2i-3} + & \frac{12}{10} \gamma_{l,2i-1} = \alpha_{l,2i-1} - t_{l+1,2(2i-1)} + \frac{1}{2} t_{l+1,2(2i-2)} 
\end{align*} \]

(3.14a)

(3.14b)

(3.14c)

Obtaining the forward transform by simply reverting the steps is impossible, since there is no rule to initialize the temporary values \( t_{l,k} \) from the \( \alpha_{l,i} \). Instead, one needs to solve (3.13) for the \( \gamma_{l,i} \). This gives – as a consequence of the overlapping supports of neighboring prewavelets – the tridiagonal SLE with constant coefficients (3.14) for level \( l \). During computation of the transform, the SLE on each level \( l \) is most efficiently solved by applying an a priori LU decomposition and only running forward and backward substitutions in \( O(2^l) \) (i.e., linear) time. The \( t_{l,k} \) again propagate information between the levels, and they again determine the bottom up direction of computations.

Note finally, other works such as [58, 37, 38] use a generating system in their derivations. The concept is adopted from the multigrid corner, where it is profitably employed in
3.2. The Core Tasks

explanations and implementations (e.g., see [54, 53]). Here, references to generating systems are deliberately avoided for clarity. Nevertheless, the connections are obvious: The $t_{l,k}$ directly correspond to nodes $(l,i), i = 2, 4, \ldots, 2^l - 2$ that appear in the generating system but not in the hierarchical basis. Operator (3.12) resolves the redundancy introduced by these nodes when switching back to the basis representation.

3.2.2. Evaluation

The task of evaluating the sparse grid interpolant $\hat{u}$ of function $u$ at some point $\vec{x}$ is an extremely simple one – at least when judged by its formal description:

$$\hat{u}(\vec{x}) = \sum_{\vec{l},\vec{i} \in P} a_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}), \quad x \in \bar{\Omega}$$

(3.15)

The naïve approach suggested by (3.15) evaluates all basis functions $\phi_{\vec{l},\vec{i}}$ at point $\vec{x}$ and creates a weighted total using the surpluses $a_{\vec{l},\vec{i}}$. Of course, the large part of $\phi_{\vec{l},\vec{i}}$ therein evaluates to zero, and the induced computational overhead is immense. After all, reading the surpluses triggers (possibly scattered) memory accesses, and the cost of tensor product evaluation of $d$-linear hats (cf. (2.4), (2.5), and (2.18)) grows linearly in $d$, while it grows on top linearly in the degree $p$ (i.e., the level $l = p - 1$) of higher order polynomials (cf. (2.38) and (2.40)). Under these circumstances, it is hardly surprising that this seemingly easy task can quickly become the computational hotspot of applications with heavy dependence on interpolated values.

A more elaborate approach is thus called for, one that reduces the complexity of the algorithm especially for large grids. Apparently, only one $\phi_{\vec{l},\vec{i}}$ per subspace is affected in an evaluation, i.e., only one basis function per subspace contributes to the final result. This is at least true for bases with non-overlapping functions, which excludes the prewavelets that are not designed for interpolation anyway. Figure 3.3a reveals the dependency patterns arising from the evaluation of such bases, and it shows how to make use of these patterns for optimized tensor product evaluation. The arrows describe the dual recursive descent into dimension and level, and since they point to perpendicular directions, partial tensor products $\prod_{j \in D'} \phi(x_j), D' \subset D$ can be reused during the descent into any of the remaining $d - |D'|$ dimensions. In Fig. 3.3a, $D' = \{1\}$ holds, i.e., $\phi_{l,i}(y)$ remains unchanged while following the green arrows along the $x$-axis. This incremental approach can lead to tremendous savings regarding 1-D basis function evaluations as can be seen in the example in Fig. 3.4.

In contrast to the transformations discussed previously, evaluation only needs to visit each affected basis function once instead of $d$ times. However, some kind of loop is still required for the computation of the tensor products as well as for the recursive descent into the dimensions. The 1-D evaluation operators sketched in Fig. 3.3b encapsulate the necessary logic in dimension independent formulations for the hat function and the higher order polynomial basis. It surprises that evaluating higher order polynomials
3. Co-Design, Part I: Formulating the Challenges

(a) evaluation in 2-D

(b) evaluation in 1-D

Figure 3.3.: In Fig. 3.3a, the affected $\phi_{l,i}$ and their hierarchical relations are highlighted for the evaluation of a 2-D DASG at point $\vec{x} = (0.625, 0.125)$. The path suggested by the arrows supports an incremental scheme for the computation of the tensor products as illustrated in Fig. 3.4. The steps shown in Fig. 3.3b can be considered 1-D evaluation operators for the hat function basis (left) and the polynomial basis (right). Except from the accumulation of result $r$, the steps are applicable in each dimension and describe how to descend to the next level in constant time. The $\eta_{l,i}$ are the precomputable normalization factors defined in (2.39).
3.2. The Core Tasks

(a) naïve scheme  
(b) incremental scheme

Figure 3.4.: The naïve computation of the tensor products in Fig. 3.4a is visually compared with the more elaborate incremental approach in Fig. 3.4b. For a level 3 grid in 3-D, the number of 1-D basis function evaluations already decreases from 30 to 19 when using the incremental scheme. In both illustrations, boxes symbolize parentheses, classical addition, and multiplication.

costs basically the same as evaluating hat functions when done in a level-wise fashion as demonstrated in Fig. 3.3b. At the same time, it is clear that the choice of a suitable data format is paramount, if memory latency is to be avoided as bottleneck in the accessing of affected coefficients. A quick peek ahead at the most common solutions for data structures in Sect. 3.3 (especially the ones in Sect. 3.3.5) can help to get an overview of available choices.

3.2.3. **UpDown – The Unidirectional Principle**

The finite element discretization of partial differential equations (PDE) (find more details in Sect. 5.3) is of central interest in sparse grid research, being for instance also the topic of Zenger’s groundbreaking work [132]. Other works building on these early findings soon followed (e.g., see [16, 17]), but the most momentous development in this line of research was without doubt the discovery of the **unidirectional principle**. This sophisticated scheme is first presented in [7, 8] as a powerful matrix-free method for the fast application of mass and stiffness matrices in sparse grid finite element contexts. Its applicability is extended for instance in [35, 19], and the principle was recently generalized further in [131, 56].

In this section, I first look at the **UpDown** scheme that forms the generic basis of the unidirectional principle. With one eye on complexities, I then analyze the benefits of basis changes in the **UpDown** computation of two operators frequently occurring in FEM
3. Co-Design, Part I: Formulating the Challenges

contexts.

Operator Splitting into Up and Down

The first question to answer is when 1-D operators require splitting into so-called Up and Down parts. To this end, I revisit the findings from [131, 56] without losing time to introduce the notation therein. Explanations build qualitatively upon the structure of matrices arising from the linear 1-D operators under examination. Readers interested in a more formal discussion or proofs are referred to the works mentioned.

Let in the following \( \text{idx} \) be a bijective mapping between a \( d \)-dimensional grid’s points and the vector component index, such that \( \text{level}(p) < \text{level}(p') \Rightarrow \text{idx}(p) < \text{idx}(p') \) for each two grid points \( p, p' \) (the order within levels is unimportant). Let \( A \) further be a 1-D operator in matrix form whose application to a grid of \( N \) points along dimension \( j \in \mathcal{D} \) is given by matrix \( A(j) \in \mathbb{R}^{N \times N} \). Consider the following splitting of \( A \) (and analogously \( A(j) \)):

\[
A = A^\Delta + A^\nabla
\]

such that

\[
A^\Delta_{ij} := \begin{cases} A_{ij} & \text{for } \text{level}(\text{idx}^{-1}(i)) \leq \text{level}(\text{idx}^{-1}(j)) \\ 0 & \text{else} \end{cases}
\]

and

\[
A^\nabla_{ij} := \begin{cases} A_{ij} & \text{for } \text{level}(\text{idx}^{-1}(i)) > \text{level}(\text{idx}^{-1}(j)) \\ 0 & \text{else} \end{cases}
\]

\( A^\Delta \) encodes dependencies of grid points on their descendants and siblings (points on same level), i.e., information is propagated up in the hierarchy. \( A^\nabla \) couples grid points with their ancestors, i.e., information from coarser levels is handed down in the hierarchy. A common denomination of \( A^\Delta \) is “Up operator”, whereas \( A^\nabla \) is often referred to as the “Down operator” of \( A \). The operator naming coincides with the matrices’ rough structure: \( A^\Delta \) resembles an upper triangular matrix, \( A^\nabla \) is a lower triangular matrix.

Let \( \mathcal{L}_L \) be a DASG given by \( \tilde{\mathcal{L}} = \{ \tilde{I} \}, \tilde{I} \in \mathbb{N}_d^+, \) i.e., \( \mathcal{L}_L \) is essentially a hierarchically decomposed full grid approximating in \( V_I \). Let \( \tilde{u} \in \mathbb{R}^N \) be a coefficient vector associated with \( \mathcal{L}_L \), and let \( \tilde{v} \in \mathbb{R}^N \) denote the result produced by proper application of \( A \) to \( \tilde{u} \) with respect to all dimensions. According to the unidirectional principle, the commutative operators \( A(j) \) can be applied to the full grid \( \mathcal{L}_L \) in the following way:

\[
\tilde{v} = \left[ \prod_{j \in \mathcal{D}} A(j) \right] \cdot \tilde{u} = A(\mathcal{D}) \cdot \tilde{u},
\]

with \( A(\mathcal{D}) \in \mathbb{R}^{N \times N} \) being the matrix form of \( A \)’s application in all \( d \) dimensions. With
3.2. The Core Tasks

(a) view on points

(b) view on subspaces

Figure 3.5.: For a full grid, the \( A_{(j)} \) can be applied in arbitrary order, as any two grid points \( p_0, p_1 \) can exchange information via common ancestors (e.g., the root) and common descendants (here: \( p_2 \)). However, if such a descendant \( p_2 \) falls victim to sparse grid truncation, the latter link is destroyed, forcing \( p_0 \) and \( p_1 \) to communicate through common ancestors. As indicated in Fig. 3.5a, this is achieved by applying \( U_p \) before \( D_p \) according to (3.18c). Figure 3.5b grants a look at data propagation on the subspace level, and it becomes clear why full grids can do without the splitting.

(3.16) and with \( 2^D \) denoting the power set of \( D \), (3.17) can be rewritten as

\[
\vec{v} = \left[ \prod_{j=0}^{d-1} \left( A_{(j)}^\Delta + A_{(j)}^\nabla \right) \right] \cdot \vec{u} \tag{3.18a}
\]

\[
= \left[ \prod_{j=1}^{d-1} \left( A_{(j)}^\Delta + A_{(j)}^\nabla \right) \right] \cdot A_0^\Delta \cdot \vec{u} + A_0^\nabla \cdot \left[ \prod_{j=1}^{d-1} \left( A_{(j)}^\Delta + A_{(j)}^\nabla \right) \right] \cdot \vec{u} \tag{3.18b}
\]

\[
= \sum_{D^\Delta \in 2^D} \left[ \prod_{j \notin D^\Delta} A_{(j)}^\nabla \right] \cdot \left[ \prod_{j \in D^\Delta} A_{(j)}^\Delta \right] \cdot \vec{u}. \tag{3.18c}
\]

Evaluation of the resulting formula (3.18) is much less straightforward than evaluating (3.17), however, it is the only option when dropping the full grid assumption and dealing with general sparse grids. The reason is explained graphically in Fig. 3.5 at the example of two points \( p_0 \) and \( p_1 \), whose basis functions have overlapping supports but no direct hierarchical relationship. During the application of the \( A_{(j)} \), information is exchanged between \( p_0 \) and \( p_1 \) through common hierarchical ancestors and descendants. However, sparse grid truncation sometimes discards the therefor needed common descendant \( p_2 \). Applying all \( U_p \) operators before \( D_p \) operators according to (3.18c) solves this problem, as data is then properly exchanged through common ancestors only.

The unidirectional principle for the general sparse grid case is usually realized via the elegant \( U_p D_p \) scheme explained in Fig. 3.6a. The recursive rule is directly derived from (3.18a) and (3.18b). Its execution triggers a total of \( 2(2^d - 1) \) 1-D operator ap-
3. Co-Design, Part I: Formulating the Challenges

(a) normal: \(2(2^d - 1)\) 1-D operators applied

(b) degenerated: \(d\) 1-D operators applied

Figure 3.6.: Figure 3.6a illustrates the common recursive UpDown scheme that implements (3.18c). In \(d\)-D, the top-level call UpDown\((d - 1)\) leads to a binary call tree with \(2(2^d - 1)\) branches and depth \(d\). Figure 3.6b shows the desirable case, in which either Up or Down is sufficient for applying all modifications to the data. As no branching or copying is necessary, the scheme degenerates to a loop over the dimensions.

Applications \((A^\Delta_{(j)} \text{ and } A^\nabla_{(j)} \text{ each counting as one})\). Furthermore, it duplicates the input data in each level of the recursion, which leads to a peak allocation of \(d\) simultaneous copies. However, an empty Down or Up operator reduces the overall work considerably (cf. Fig. 3.6b). The whole scheme then degenerates to a linear sequence of \(d\) applications of the non-empty operator, which is exactly what is observed for the basis transformations in Sect. 3.2.1. The hierarchization operators, for instance, can both be expressed as \(A^\nabla\) matrices, in which points only depend on their ancestors. The following section contains a discussion of more complex tasks, for which the application of the full UpDown scheme can generally not be avoided.

Application of Mass and Stiffness Matrices via UpDown

In most FEM contexts, the continuous PDE description of a system is discretized in a system of linear equations (SLE) which is then solved with an iterative SLE solver. A frequently occurring subtask of this process is the application of so-called mass and stiffness matrices to vectors, the very task for which the UpDown scheme was originally developed. The motivation comes from the drawbacks of the alternative, the explicit assembling of the matrices: Although the number of unknowns is usually much smaller than in non-hierarchical FEM settings, overlapping hierarchical basis functions cause much fill-in and lead to a non-sparse (but also not fully dense) matrix structure. The unidirectional principle on the other side promises – in conjunction with efficient linear runtime Up and Down operators – the application of such stiffness matrices to vectors.
3.2. The Core Tasks

in only linear time\(^1\). This is remarkable considering the quadratic time needed for the application of a non-sparse matrix. The following paragraphs demonstrate how this is possible at the example of two common operators.

Consider a grid of \(N\) points and the mapping \(idx\) from before. Solving via FEM involves the application of matrices arising from bilinear forms of the basis functions \(\phi_j, j \in \{0, \ldots, N - 1\}\), such as \(m\) for the \(L_2\) scalar product and \(l\) for the Laplacian:

\[
m(u, v) = \int_{\Omega} u \cdot v \, d\Omega = \langle u, v \rangle, \quad (3.19)
\]

\[
l(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega = \langle \nabla u, \nabla v \rangle. \quad (3.20)
\]

The resulting matrix forms \(M_{\phi(D)}\) and \(L_{\phi(D)}\) for the \(d\)-dimensional case are given by

\[
M_{\phi(D)} := (m_{i,j}) \in \mathbb{R}^{N \times N}, \quad m_{i,j} := \langle \phi_i, \phi_j \rangle = \prod_{m=0}^{d-1} \langle \phi_{i_m}, \phi_{j_m} \rangle, \quad (3.21)
\]

\[
L_{\phi(D)} := (l_{i,j}) \in \mathbb{R}^{N \times N}, \quad l_{i,j} := \sum_{k=0}^{d-1} \left\langle \frac{\partial \phi_{i_k}}{\partial z_k}, \frac{\partial \phi_{j_k}}{\partial z_k} \right\rangle \cdot \prod_{m \neq k} \langle \phi_{i_m}, \phi_{j_m} \rangle, \quad (3.22)
\]

and the unidirectional principle is used in the following to efficiently compute them and apply them to a vector on the fly.

The 1-D Up and Down operators for \((3.21)\) with respect to the hat function basis are given in their recursive forms in Alg. 2. Quite similar to the hierarchization operator \((3.7)\), they rely on clever combinations of hierarchical piecewise linear functions for constant time computation of integral sums (for a detailed derivation see for instance [106, 107]). Consequentially, both operators need one read and one write access per coefficient and have linear runtime. The 1-D operator for the energy scalar product of hat functions corresponds to the diagonal scaling \((3.23)\), and its obvious realization as Up operator upHatL2Energy (with empty Down part) is given at the bottom of Alg. 2. Together with downHatL2 and upHatL2, it completes the set of operators needed for the application of the Laplacian bilinear form \((3.22)\).

\[
\langle \phi_{l,i}^l, \phi_{l',i'}^l \rangle = \begin{cases} 2^{l+1} & \text{for } (l, i) = (l', i') \\ 0 & \text{else} \end{cases} \quad (3.23)
\]

Regardless of the undoubted elegance and conciseness of the approach, the long sequences of 1-D operators \(2(2^d - 1)\) for \(M_{\phi(D)}\), \(d(2^d - 1)\) for \(L_{\phi(D)}\) typically entailed by the unidirectional principle are still undesirable. Among other things, wavelets have been

\(^1\)In the sparse grid context, “linear time” typically neglects “constant” factors involving the dimensionality \(d\). E.g., the UpDown scheme for applying \(M_{\phi(D)}\) triggers \(2(2^d - 1)\) 1-D operator applications, while basis transformations usually do with only \(d\) 1-D operator applications. Since this work is among other things about the tuning of algorithms, such factors are from now on specified explicitly for better comparison.
Algorithm 2 The function $\text{downHatL2}$ computes for each $\phi_{l,i}$ in the 1-D hierarchy the summed up coupling with all its ancestors $\sum_{l'=1}^{l-1} \sum_{i' \in I_{l'}} \langle \phi_{l,i}, \alpha_{l',i'} \cdot \phi_{l',i'} \rangle$. Dehierarchization during the recursive descent allows to evaluate these sums in constant time, leading to overall linear time. Since $M$ is a symmetric matrix, $\text{upHatL2}$ implements the transposed algorithm and computes sums $\sum_{l'=1}^{n} \sum_{i' \in I_{l'}} \langle \phi_{l,i}, \alpha_{l',i} \cdot \phi_{l',i} \rangle$ (note the additional coupling with level $l$). The matrix for the energy scalar only has an entry on the diagonal, i.e., only the $\text{Up}$ operator $\text{upHatL2Energy}$ is needed.

```
1: function downHatL2(l, i, u_l, u_r, n, \{\alpha_{l,i}\}, \{m_{l,i}\})
2:   u_m \leftarrow (u_l + u_r)/2 \triangleright for dehierarchization: compute mean value
3:   m_{l,i} \leftarrow u_m \cdot 2^{-i} \triangleright accumulate $\sum_{l'=1}^{l-1} \sum_{i' \in I_{l'}} \langle \phi_{l,i}, \alpha_{l',i'} \cdot \phi_{l',i'} \rangle$
4:   if $l < n$ then \triangleright not on highest level? descend recursively
5:      downHatL2(l + 1, 2i - 1, u_l, u_m + \alpha_{l,i}, n, \{\alpha_{l,i}\}, \{m_{l,i}\})
6:      downHatL2(l + 1, 2i + 1, u_m + \alpha_{l,i}, u_r, n, \{\alpha_{l,i}\}, \{m_{l,i}\})
7:   end if
8: end function
9: :
10: function upHatL2(l, i, &u_l, &u_r, n, \{\alpha_{l,i}\}, \{m_{l,i}\})
11:   u_p \leftarrow 0, u_{p'} \leftarrow 0 \triangleright implicit in transposed: finalizing initialization
12:   if $l < n$ then \triangleright not on highest level? descend recursively
13:      upHatL2(l + 1, 2i - 1, u_l, u_p, n, \{\alpha_{l,i}\}, \{m_{l,i}\})
14:      upHatL2(l + 1, 2i + 1, u_r, u_{p'}, n, \{\alpha_{l,i}\}, \{m_{l,i}\})
15:   end if
16:   m_{l,i} \leftarrow u_p + u_{p'} + \alpha_{l,i} \cdot 2^{1-l}/3 \triangleright accumulate $\sum_{l'=1}^{n} \sum_{i' \in I_{l'}} \langle \phi_{l,i}, \alpha_{l',i'} \cdot \phi_{l',i'} \rangle$
17:   u_l \leftarrow u_l + (u_p + u_{p'})/2 + \alpha_{l,i} \cdot 2^{-l-1}
18:   u_r \leftarrow u_r + (u_p + u_{p'})/2 + \alpha_{l,i} \cdot 2^{-l-1}
19: end function
20: :
21: function upHatL2Energy(n, \{\alpha_{l,i}\}, \{g_{l,i}\})
22:   for all $1 \leq l \leq n, i \in I_l$ do
23:      g_{l,i} \leftarrow \alpha_{l,i} \cdot 2^{l+1} \triangleright compute $\langle \phi_{l,i}, \alpha_{l,i} \cdot \phi_{l,i}' \rangle$
24:   end for
25: end function
```
3.2. The Core Tasks
designed with a reduction of this cost in mind. The postulated orthogonality between
the increments $W_l$ leads to the following sparse coupling between interior basis functions
(boundaries are discussed later in this section):

$$\langle \psi_{l,i}, \psi_{l',i'} \rangle = \begin{cases} 
1/3 & \text{for } l = l' = 1 \\
1/25 & \text{for } l = l' = 2, i \neq i' \\
44 \cdot h_l/75 & \text{for } (l, i) = (l', i'), i \in B_l \\
18 \cdot h_l/25 & \text{for } (l, i) = (l', i'), i \in B_l \\
11 \cdot h_l/75 & \text{for } l = l', |i - i'| = 2, i \in B_l \cup i' \in B_l' \\
2 \cdot h_l/15 & \text{for } l = l', |i - i'| = 2 \\
-1 \cdot h_l/75 & \text{for } l = l', |i - i'| = 4 \\
0 & \text{else,}
\end{cases}
$$

(3.24)

where $h_l = 2^{-l}$ is the mesh width and $B_l = \{1, 2^l - 1\}$ the index set of near boundary
functions. A pentadiagonal matrix $M_{\psi}(D)$ (as a pendant to $M_{\phi}(D)$ in (3.21)) arises for
the $L_2$ scalar product (3.24), whose realization takes a linear runtime $Up$ operator $M_{\hat{\psi}}$.
Consider the following relation between a stiffness matrix $M$, a transformation matrix $H$
and its transposed operator $H^T$:

$$M' = H^T M H \quad \iff \quad M = H^{-T} M' H^{-1}.
$$

(3.25)

Relation (3.25) plays a role for instance in the multigrid context (see [2]), where different
function representations are sometimes of advantage for the application of smoothers.
Let $H$ for now be the transformation matrix from the prewavelet to the hat function
basis (cf. (3.13)). Since $H = H^\Delta$ holds, $M_{\phi}(D)$ can be rewritten (and applied) as

$$M_{\phi}(D) = \sum_{D^{\Delta} \in 2^D} \left( \prod_{j \in D^{\Delta}} M^\nabla_{\phi}(j) \right) \cdot \left( \prod_{j \in D^{\Delta}} M^\Delta_{\phi}(j) \right)
$$

(3.26a)

$$= H_{(D)}^T \cdot M_{\psi}(D) \cdot H_{(D)}^{-1}
$$

(3.26b)

$$= \left( \prod_{j \in D} H_{(j)}^\Delta \right)^{-1} \cdot \left( \prod_{j \in D} M^\Delta_{\psi}(j) \right) \cdot \left( \prod_{j \in D} H_{(j)}^\Delta \right)^{-1}
$$

(3.26c)

Switching from (3.26a) to (3.26c) implies applying $2(2^d - 1) - 3d$ 1-D operators, which
already pays off for $d = 3$. All extra work required is the derivation of the transposed
transformation operator $H^{-T}$ from $H^{-1}\Delta$, leading to a Down operator (with diagonal).
Let now $E_{\phi} = E_{\phi}^\Delta$ be the operator for the energy scalar product (3.23). The pendant for
the prewavelet basis $E_{\psi}$ poses a challenge, as the wide function supports cause massive
matrix fill-in, ruling out a straightforward linear runtime implementation and making an
UpDown splitting necessary. Instead, providing $H^T = (H^\Delta)^T$ and temporarily switching
back to the hat function basis permits the reuse of $E_{\phi}^\Delta$ in an optimized application of
3. Co-Design, Part I: Formulating the Challenges

the Laplacian $L_{\phi}(D)$ \([3.22]\), as is described in \([3.27d]\):

$$
L_{\phi}(D) = \sum_{D^\Delta \in 2^D} \left( \prod_{j \in D^\Delta} L^\nabla_{\phi(j)} \right) \cdot \left( \prod_{j \in D^\Delta} L^\Delta_{\phi(j)} \right)
$$

\hspace{1cm} (3.27a)

$$
= H^{-T}_{(D)} \cdot L_{\psi}(D) \cdot H^{-1}_{(D)}
$$

\hspace{1cm} (3.27b)

$$
= \left( \prod_{j \in D} H^\Delta_{(j)} \right)^{-T} \left( \sum_{k \in D} \left[ E_{\psi(k)} \cdot \prod_{k \in D \setminus k} M_{\psi(j)} \right] \right) \left( \prod_{j \in D} H^\Delta_{(j)} \right)^{-1}
$$

\hspace{1cm} (3.27c)

$$
d(d+2) \text{ 1-D operator applications}
$$

\hspace{1cm} (3.27d)

Stepping from \([3.27a]\) to \([3.27d]\) implies saving $d(2^d - 1) - d(d+4)$ operator applications, which leads to an obvious benefit for $d \geq 4$. However, all 1-D operators in \([3.27d]\) have linear runtime, and as this is not the case for the $L^\Delta_{\phi(j)}$ and $L^\nabla_{\psi(j)}$ in \([3.27a]\), a positive effect should be noticed even for smaller $d$.

Now, a few more words about the prewavelet basis are due. In combination with basis changes, it can obviously be used to reduce the runtime complexity of algorithms for the linear hat function basis. But of course, PDEs can also directly be solved in the prewavelet space. For one, it removes the need for the bracketing basis transformations around $M_{\psi}(D)$ and $L_{\psi}(D)$ in \([3.26c]\) and \([3.27d]\). But more importantly, $M_{\psi}(D)$ and $L_{\psi}(D)$ usually lead to better condition numbers in the system matrix than $M_{\phi}(D)$ and $L_{\phi}(D)$, which makes prewavelets so popular for preconditioning (cf. convergence in Sect. 5.3.5). When explicit boundaries are used, the latter even becomes the prewavelets’ only remaining advantage. The reason are non-empty couplings between boundary and near-boundary functions (cf. \([3.28]\)), which make full-grown UpDown schemes also necessary for the application of $M_{\psi}(D)$ and $L_{\psi}(D)$. The scalar product of these functions at the left boundary assumes values

$$
\langle \psi_{0,0}, \psi_{l,i} \rangle = \begin{cases} 
1/3 & \text{for } (l, i) = (0, 0) \\
1/6 & \text{for } (l, i) = (0, 1) \\
1/4 & \text{for } (l, i) = (1, 1) \\
2 \cdot h_l/5 & \text{for } i = 1 \\
0 & \text{else,}
\end{cases}
$$

\hspace{1cm} (3.28)

which also hold for the couplings between $\psi_{0,1}$ and $\psi_{l,2^l-1}, 0 < l \leq n$ at the right boundary due to symmetry.
3.3. Commonly Used Sparse Grid Data Structures

3.2.4. Summary of Observations

The assessment of the tensor product’s implications in Sect. 3.1 is completely validated by the subsequent study of actual sparse grid tasks. The analysis of algorithms affirms that the efficiency of the 1-D operators and of their parallel application to a grid must be given highest priority in a fast implementation. The unveiling of basis transformations as a special case of the general unidirectional principle underpins this insight, as it is now clear that the same set of optimization criteria serves both cases. A related key observation is that investing in clever basis changes can offer huge benefits. Under certain circumstances, they for instance allow to avoid the UpDown splitting even for complex operators, thus enabling these operators for special treatment in $O(d)$ instead of the common $O(2^d)$ computational passes. Furthermore, basis changes sometimes lead to more favorable problem formulations and open the door for more intuitive and straightforward treatment.

The examination of 1-D operator application leads to two perspectives: the view on the data and the view on the whole sparse grid. On the data level, the structure of the unidirectional principle indicates that the smallest scope for localized processing is the scope of 1-D trees. Since 1-D operators potentially yield full data dependency graphs within these trees, their fast application urgently requires low overhead access to all coefficients in the 1-D hierarchy. From the sparse grid’s point of view, the application of a 1-D operator reminds of localized stencil applications to independent subsets of the data. This calls for efficient (parallel) traversal schemes, and the uniform shape exhibited by many 1-D trees motivates simultaneous vectorized processing of these trees.

Optimized sparse grid evaluation is subject to different considerations, also because the amount of work is determined primarily by the number of evaluations, while the grid’s size ranks only second. It is obvious how the task of evaluating the grid at many different points can be accomplished in an embarrassingly parallel fashion in TLP or even NLP implementations. It is, however, less obvious how SIMD processing can be exploited, since invocations for differing evaluation points generally entail different data access patterns and different sets of affected coefficients. A pressing issue is also the cost balance between arithmetic and memory operations, especially if the aim is to hide memory latency behind computation. Simply ensuring optimal algorithmic complexity is then insufficient (e.g., through clever reuse of partial tensor products), as memory quickly turns into an implementation’s bottleneck if the data structure exhibits poor data locality for the affected coefficients.

3.3. Commonly Used Sparse Grid Data Structures

The final part of the analysis comprises the examination and comparison of data structures encountered in publications and sparse grid codes. With the knowledge from the
previous section about common tasks on sparse grids, it is possible to immediately recognize strengths and weaknesses of one storage form or the other. I match these properties with the previously stated requirements and probe the data structures’ fitness regarding the usual tasks without conducting experiments. I begin the overview with storage forms for the most general SASG, before I turn to increasingly more restrictive DASG, truncated and regular grids. With this order, it shall become clear which simplifying assumptions can be made for more restrictive grid types, and which optimizations are facilitated by these assumptions.

I must point out that when it comes to sparse grid data structures, the literature is again extremely sparse. The vast majority of sparse-grid-related publications never touch the topic of data structures let alone describe the implementation in detail that was used to produce presented results. I can therefore not rule out that there may be other sophisticated solutions beside the ones discussed here, as such solutions could have easily slipped past me. Note further that I give references where I can for what is described in this section. However, some statements may still miss a reference as the knowledge has either been informally exchanged through private communication, or the insight originates from unpublished (and undocumented) self-conducted experiments. I include such information where I find that it enriches the discussion, but be assured that I do not jump to conclusions based on unreliable information.

### 3.3.1. Tree-Like Structures

I choose trees in the introductory explanations in Sect. 2.2.1 because they come to mind automatically when thinking about sparse grids. The topology of tree nodes can directly mirror the hierarchical relations between grid points, which grants intuitive access to the notions of levels, adaptivity, and refinement. As evidenced by the references given, trees make their appearance mainly in early sparse grid implementations and are hardly given any attention lately. I will in the following mention a few relevant variants.

**Directed Graphs**

A sparse grid’s self-similarity enables a very concise representation via pointers. Each grid point is root to its own subtree, which makes constructing a classical $d$-dimensional SASG straightforward: A simple grid point class can encapsulate $2d$ references to a point’s potential children, as well as another reference to the point’s associated data (it can also directly store the data). With a perfectly compact data layout and 8 bytes per pointer and data item (e.g., a scalar double precision value), the storage consumption of a grid with $N$ points amounts to around $16dN$ bytes.

Of course, the overhead can be lowered by removing the redundancy from the set of links; in fact, the recursive construction visualized in Fig. 2.7 generates exactly such
3.3. Commonly Used Sparse Grid Data Structures

Figure 3.7.: The center of the image shows the pointer structure of a 1-D tree (top) and a 2-D tree’s recursive construction from 1-D trees (bottom). On the left, the indices 0 to 19 of the heterogeneous array are shown, in which the 1-D tree from the center is serialized following a depth first traversal (dashed blue line) according to the scheme presented in [116]. $s$ marks a size entry for the next subtree (measured in array entries), and the boolean $l$ ($r$) is set if a left (right) subtree exists. The array on the right demonstrates the recursive continuation of the principle for the 2-D grid from the center. Dots indicate where serializations of 1-D trees are left out of the illustration for clarity.
3. Co-Design, Part I: Formulating the Challenges

minimalistic trees. A more pragmatic construction rule could be as follows: For building a regular \(d\)-dimensional level \(n\) sparse grid, the process initially creates a 1-D tree of level \(n\) along dimension \(d - 1\). It then recursively spawns off a 1-D tree of level \(n - l\) along dimension \(d - 2\) for each tree node, where \(l\) is each node’s global level. This recursive process stops with the completion of dimension 0. The center of Fig. 3.7 illustrates the step from 1-D to 2-D for a sparse grid with boundaries. Implementations based on this principle are for example used in [17, 7], but concrete numbers regarding storage consumption are not given in these works.

Serialized Tree Representations

Not all tree-based variants use pointers. In [116, 2], a solution is explained that serializes a dimension-recursive depth first traversal of an SASG and stores it in a linear array. The resulting heterogeneous array representation interlaces the grid coefficients with the structural information directing the traversal. A derivation and measurements for the cost of this representation is found in [116], and it amounts to around \(3N\) array entries for the structural information plus \(N\) entries for the grid data. Figure 3.7 explains the details of this serialization for the 1-D and 2-D case, but the knowledge can easily be transferred to \(d\) dimensions.

Apparently, a one to one mapping between the pointer-based trees and the serialized arrays exists, however, the latter have advantages regarding data locality. Favoring caches, all 1-D subtrees are compact in memory, and as the format is endowed with the sizes of subtrees, fast navigation, copying, and extraction of substructures is supported. Furthermore, the self-contained flat array representation can be ported to any platform and programming model. Note that another data structure based on a serialized tree-representation is examined in Sect. 3.3.5. It is included in the discussion there, because it lacks support for SASG, plus, it rather fits the other data structure category due to its dominating properties.

I close the discussion of tree-based storage forms for now by pointing out an important property shared by all members of this category: Direct information about a grid point’s level or position is stripped from all representations for efficiency. It is left to the visiting algorithm to carry and update grid point information consistently. However, the unavailability of such information in the grid rules out random access, and it forces algorithms to start their (often recursive) traversal at the root node. This constraint considerably complicates effective parallelized processing, an aspect simply not as relevant in times when these data structures were still fashionable. Above all, the application of 1-D operators is in case of the more space-efficient trees only favored along the very first dimension.
3.3. Commonly Used Sparse Grid Data Structures

Figure 3.8.: Point \((l, i)\) is identified by a bit sequence of dynamic length which encodes the path from \((1, 1)\) to \((l, i)\) in the 1-D binary tree. In principle, \(l - 1\) bits are sufficient to uniquely represent \((l, i)\), but in \(d\) dimensions concatenations of \(d\) such sequences need to be interpreted unambiguously. Therefore, a structural ‘1’ bit precedes each relevant bit (underlined), and one ‘0’ bit separates the entries in the concatenation. The 3-D point \((2, 1, 3), (1, 1, 3)\) thus leads to the bit string "10[0][0]1011", where the boxed zeros separate the three components.

3.3.2. Hash Maps for Spatial Adaptivity

Hash tables (also called hash maps) are first employed to express irregular structures in adaptive PDE codes in the late 1990s. Early works are found in both general scientific computing (e.g., see [60]) and in the context of sparse grids (see [55, 18]). For sparse grid codes, the most charming properties of hash tables consist in

- the promise of random access into arbitrary SASG in \(O(1)\) time,
- a uniform interface to all tree nodes directly based on multi-indices,
- easy maintenance of changing grids, and
- a straightforward way to separate the grid data from the data structure.

For all these reasons, hash tables are to this day the most widely spread data structure for SASG, used in educational codes (see [46]) and high performance codes (see SG++ [107]) alike. Compared to other data structures for sparse grids, they have been studied in some more detail, which further signifies their importance and popularity. Some of the more involved examinations can be found in [55, 115, 114, 34], and more recently the topic has been picked up again in [37, 38, 107].

Published during the dawn of the multi-core era, the latter works have also started to promote the discussion of hash maps’ fitness for high performance applications on modern parallel platforms. To this end, different forms of hash keys and hash functions have been the subject of studies. Here, I only mean to scratch the surface of this topic, and so I do with very briefly mentioning three general categories of hash keys:
1.) Some variants encode the $2d$ integer entries of the level-index vector pair in a (raw) string or map them (randomly) to an integral type (e.g., a 64-bit unsigned integer). Both key types are accepted by contemporary hash map implementations such as the one found in the C++ Standard Template Library\(^2\) (STL). Since an unordered map (most commonly used form, see \([55, 114, 107]\)) performs best when the hashing function distributes the keys across all bins equally, the “random approach” leads to quite satisfying results and permits flexible annotation of the keys with additional information such as flags for leaf grid points. Proof of this success is the SG++ project \([107]\), in which such a strategy is pursued.

2.) A more target-oriented handling of keys is sought in \([34]\). In order to be able to manipulate keys directly for fast navigation in the grid hierarchy, the author uses a bitwise encoding scheme for a compact representation of the level-index vector pair as shown for non-boundary grids in Fig. 3.8. The cost for compressing $(\vec{l}, \vec{i})$ on level $n$ in $d$ dimensions amounts to

$$2n + d - 3 = 2 \cdot |\vec{l}|_1 - d - 1 \text{ bits,} \quad (3.29)$$

which allows to handle keys for moderate dimensionality and level in a 64-bit integer. An extension of the format supporting grids with boundary points is not mentioned in \([34]\), but I believe at least $2d$ additional bits need to be spent.

3.) The third approach relies on explicit enumeration schemes, which assign each grid point a unique integer index. Schemes with dense index ranges have been found for regular \([37, 94, 78]\) and even truncated sparse grids \([93, 25]\) (both are discussed in the upcoming Sect. 3.3.5), but adaptivity opens holes in the ranges and rules out a direct mapping between the grid points of an SASG and a consecutive array index. Nevertheless, the bijective mappings still deliver unique and compact integer identifiers well suited to be used as hash keys.

Time and various implementations on several generations of workstations have revealed, that the choice of the hash key only significantly impacts the performance when the quality of the distribution in the hash map is concerned. It has for example been observed in experiments mentioned in \([107]\), that for some sparse grid algorithms more than 80% of the execution time is spent in hash-map-related tasks, such as the hashing of keys, the resolution of collisions, and the loading of coefficients. Proof that especially the latter tends to be a real performance blocker in hash-map-based implementations is found in \([69]\), where SMT on all 24 logical cores of a 12 core shared memory system leads to speedups of around 19, indicating severe memory latency boundedness. I want to conclude this short mentioning of hash maps with a list of their less preferable traits. Hash maps

- have considerable storage overhead,

- lead to memory latency bounded applications despite $\mathcal{O}(1)$ time data access,

3.3. Commonly Used Sparse Grid Data Structures

• involve at least $\mathcal{O}(d)$ operations for the construction / manipulation of hash keys,
• are unsuitable for implementations on accelerator platforms,
• work best when data is randomly scattered and thus non-local,
• control every aspect of data storage, thus inhibiting explicit data alignment desirable for vectorization or general exploitation of known data access patterns.

3.3.3. Array of Structs (AoS) and Struct of Arrays (SoA)

Data structures based on simple arrays of objects are still a niche product in the sparse grid world, probably because their simplicity seems to contradict the concept of sophisticated hierarchical structures at first glance. Nevertheless, I want to point out three examples, two of which have just recently gained importance, while the third example represents an early attempt that is briefly mentioned for completeness. I begin with the latter.

BASIS3 – Interconnected Patches

BASIS3 is the name under which an early implementation of a 3-D SASG is published in a 1993 technical report [70], followed by a 1997 re-publication as book chapter [71]. The data structure only offers a limited set of capabilities which specifically meet the demands of three-dimensional computational fluid dynamics (CFD) simulations (it can, however, be used in 1-D and 2-D settings as well). The authors target an implementation in FORTRAN 77 (source code included in the publication), which rules out pointer-based trees from the set of options. The concept of patches is therefore introduced, each patch representing the cell around a grid point (i.e., the basis function’s support). Avoiding actual pointers, all patches are arranged in lists whose elements are linked by means of indices pointing to their respective neighbors, parents, and children.

Due to its obvious limitations, the data structure itself has little to offer for the kind of implementation targeted here, but a slightly shifted view on the AoS at least draws a final line under the discussion of trees: A second interpretation of this storage form is that of a tree serialized in an array of interconnected elements. BASIS3 is the only example I have been able to find where such a technique is applied to pack a tree’s nodes compactly in order to increase the data locality.
Key-Value Approach

The approach sketched in the following is so far unpublished, although the ground work was laid by Riko Jacob already in 2011 in form of a prototypical implementation of basic functionality such as adaptive refinement and linear hierarchization. At the time of its conceiving, this implementation was benchmarked against the corresponding functionality in SG++, leading to extremely promising results for sequential and especially shared memory parallel configurations. Close collaboration with R. Jacob on the topic (embodied in two joint spin-off publications [78, 25] about related non-adaptive implementations) and the long-existing agreement to publish the approach together give me the chance to elaborate on it in some more detail here. A short outline follows now, and in Sect. 5.4 the discussion continues with the presentation of a new enhanced implementation that benefits from ideas of this thesis. Explanations then include a performance comparison of the new implementation with the respective SASG functionality in SG++, revealing the potential of the approach to the public for the first time.

The underlying idea for the data structure under examination is a recursive enumeration scheme for a sparse grid’s points. Its benefits have already been exploited in very successful implementations of algorithms for regular [78] and – after minor modifications – also truncated [25] sparse grids (compare upcoming discussion in Sect. 3.3.5). The major challenge for the SASG variant consists in finding a strategy for handling the gaps spatial adaptivity introduces in the index ranges. But instead of using the index as hash key as already proposed in the previous discussion of hash maps, R. Jacob removes the hash map layer and directly takes advantage of some of the enumeration’s properties. Now, the mapping itself is explained for the no-boundary case, and the efficient handling of non-dense index ranges is demonstrated.

Performing a breadth-first search (bfs) in a (non-adaptive) binary tree while assigning its nodes consecutive indices leads to the following mapping:

$$bfs^{(1)}(l, i) := 2^{l-1} - 1 + \left\lfloor \frac{i}{2} \right\rfloor, \quad (l, i) \in \mathbb{N}_+ \times \mathbb{I}_l.$$  \hspace{1cm} (3.30)

In a process paralleling the steps of the recursive tree construction rule described in Sect. 3.3.1 the enumeration is naturally extended to multiple dimensions through recursive insertion and expansion of lower-dimensional subtrees. As can be seen in Fig. 3.9 the resulting index does not advance level-wise, and so a global constraint in form of the maximum sparse grid level \(n\) is needed for the mapping (a large value like 30 usually serves here). Let \(vol(d, l)\) denote the number of points in a \(d\)-D grid of level \(l\). The
Figure 3.9.: The consecutive multi-dimensional bfs index of a 4-D sparse grid \((n = 3)\) and its lower-dimensional constituents is shown. Each 3-D subgrid is framed by a box, in which 2-D subgrids are separated by horizontal lines. Rows correspond to 1-D subgrids, and vertical separators within rows mark level transitions. The index of the 1-D root is written before each row. Subgrid sizes can be matched with the lookup table in the lower right corner.
3. Co-Design, Part I: Formulating the Challenges

extension of the 1-D mapping \((3.30)\) to \(d\) dimensions is then given by

\[
\text{bfs}^{(d)}(\vec{l}, \vec{i}) := \sum_{t=1}^{d} \left( \text{vol}(t, n - |\vec{r}_{t,d}|) - 2^{l_{t-1}-1} \cdot \text{vol}(t, n - |\vec{r}_{t-1,d}|) + \left\lfloor \frac{i_{t-1}}{2} \right\rfloor \cdot \text{vol}(t-1, n - |\vec{r}_{t-1,d}|) \right) 
\]

(3.31)

where \(\vec{r} = (\vec{l} - 1) \in \mathbb{N}^d\) and \(\vec{r}_{j,k} = (r_j, \ldots, r_{k-1}) \in \mathbb{N}^{k-j}\). Here and subsequently, both mappings \((3.30)\) and \((3.31)\) are dynamically referred to as bfs mappings. A more detailed discussion of the \(d\)-dimensional \(O(d)\) bfs mapping and its \(O(d + n)\) inverse (which is needed in the approach, too, but lacks a concise form) is left for the planned publication.

Anyway, the correctness of \((3.31)\) is easily verified when tested against the grid shown in Fig. 3.9 (for \(n = 3\)).

The elegance of the approach stems from its simplicity: An arbitrary SASG with \(N\) points is described by a sorted vector

\[
\bigotimes_{j=0}^{N-1} (\text{bfs}^{(d)}(\vec{l}_j, \vec{i}_j), \alpha_{\vec{l}_j, \vec{i}_j}) \in (\mathbb{N}, \mathbb{R})^N, \quad \text{with } \text{bfs}^{(d)}(\vec{l}_j, \vec{i}_j) < \text{bfs}^{(d)}(\vec{l}_k, \vec{i}_k) \Rightarrow j < k, 
\]

(3.32)

where \((\vec{l}_j, \vec{i}_j)\) denotes the \(j\)-th level-index vector pair in the list (not the vectors’ \(j\)-th components). For fast index calculations according to \((3.31)\), a lookup table of size \(d \times n\) buffering the subgrid sizes \(\text{vol}(\cdot, \cdot)\) makes sense. Its entries are most efficiently determined via the recursive dependence

\[
\text{vol}(d, n) := \text{vol}(d-1, n) + 2 \cdot \text{vol}(d, n-1),
\]

(3.33)

which is already used in the point set construction in Fig. 2.6. It is due to the bfs mapping’s construction that – as long as the array is sorted – all 1-D subtrees with respect to dimension 0 reside in contiguous memory, even for non-dense SASG. The trees are thus perfectly aligned for the application of 1-D operators along this dimension. This observation motivates the central idea behind the whole approach, which is to ensure such ideal conditions for all processed dimensions by

1.) computing level and index vectors via the inverse mapping \((\vec{l}_j, \vec{i}_j) := \text{bfs}^{(d)-1}(\text{key}_j), \)
2.) cyclically shifting their components by one (e.g., \((l_0, l_1, \ldots, l_{d-1}) \rightarrow (l_1, \ldots, l_{d-1}, l_0))
3.) recomputing the bfs mappings for this order of dimensions,
4.) sorting the key-value pairs with respect to the keys,
5.) applying the 1-D operator in the new “fast” dimension,
6.) and returning to step 1.) if not yet finished.
While this sounds like a lot of work, the remapping of the keys and the sorting of the array via a fast radix sort are nicely parallelizable. Furthermore, concurrent operator application to multiple compactly stored subtrees boosts the throughput far beyond what is observed for any hash map. This is expected to be also true for grid evaluation, since the storage layout clusters affected coefficients and favors the reuse of partial tensor products as described in Sect. 3.2.2. Last but certainly not least, both the storage form and the algorithmic steps are suitable for accelerator platforms as well.

SoA for Streaming Algorithms

A few recent works set the goal to porting SASG algorithms to modern many-core architectures in order to optimally exploit these architectures’ parallel resources. The enveloping project was launched by the developers of SG++, who recognized the widening conceptual gap between the capabilities of increasingly parallel platforms and the demands of sophisticated sparse grid algorithms operating on hash maps. The efforts in the project are guided by the central assumption that up to a break-even point determined by the problem size, non-optimal algorithms may still deliver better performance than those with optimal complexity, as long as perfect utilization of the target platforms’ parallel compute units is guaranteed. The validity of the assumption is shown in publications related to data mining [69, 66] and computational finance [67, 66]. The results of these works are another indicator that the purely classical modes of discussion, which fail to take machine constraints into consideration in the analysis of algorithms, have started to grow obsolete.

The approach is best explained at the example of data mining via an SASG constructed from piecewise d-linear basis functions $\phi_j$, $0 \leq j < N$. See Sect. 5.1 for more information about why this task depends heavily on the grid’s fast evaluation at a rather large number of points. During this computationally most expensive part of the program, the grid’s contents remain unchanged, allowing the authors to temporarily abandon the grid’s hash map representation for a specifically designed SoA representation, that is further endowed with precomputed information supporting efficient evaluation. In this step, the grid structure is flattened and all knowledge about hierarchical relations between grid points is dropped. Following the naïve approach in (3.15), each basis function is then evaluated at each evaluation point using the universally applicable formula for 1-D evaluation

$$\phi_{l,i}(x) = \max \left(0, 1 - |2^l x - i|\right).$$

(3.34)

According to (3.34), evaluating one $\phi_j$ takes $5 \cdot d + O(d) + 1$ FLOPs (evaluation of 1-D parts + reduction of result + multiplication with surplus), given precomputed vectors

$$\overrightarrow{j}(N) := \prod_{j=0}^{N-1} 2^j \in \mathbb{R}^{dN}, \quad \overrightarrow{l}(N) := \prod_{j=0}^{N-1} \vec{l}_j \in \mathbb{R}^{dN}, \quad \overrightarrow{\alpha}(N) := (\alpha_0, \ldots, \alpha_{N-1}) \in \mathbb{R}^N \quad (3.35)$$

exist for streaming, and given that the absolute value of floating point numbers is determined via a blending operation. Concurrency between the single evaluation tasks is
easily achieved. But now, vectorization is on top accomplished on virtually every platform with vector register width $w$, by merely interlacing the components of $w$ consecutive vectors $\vec{l}_{kj}, \ldots, \vec{l}_{kj+w-1}$ and $\vec{i}_{kj}, \ldots, \vec{i}_{kj+w-1}, 0 \leq k < N/w$ in the definition (3.35) of $\vec{l}^{(N)}$ and $\vec{i}^{(N)}$. The resulting algorithm can unconditionally take advantage of ILP, TLP, and NLP, and it is even well suited for heterogeneous distributed systems. Note that, with some modifications, the strategy has been successfully applied to modified linear basis functions as well, but note also that it is unsuitable for polynomial basis functions which cannot be evaluated in uniform operations.

As for computational finance, a similar principle is used for the application of mass and stiffness matrices (see discussion in Sect. 3.2.3). The cost of repeated executions of the complexity-optimal $UpDown$ algorithm is weighed against a one-time assembling and storing of these (dense) matrices, which can then be reused for efficient repeated applications in the iterations of the SLE solver. Again, the uniform structure of the hat basis functions is exploited in the parallelized and vectorized calculations of the matrix entries. However, in contrast to the data mining setting, execution on a cluster becomes necessary already for tens of thousands of grid points.

Even if the presented SoA storage form hardly inspires the development of a sophisticated all-round data structure for sparse grids, the approach still teaches two important lessons: First, it can pay off to desert established complexity-optimal schemes for ones that show better compatibility with the capabilities of the target platform. Second, there is no reason to stubbornly hold on to one data structure, if another data structure obviously presents the better choice for the task at hand.

### 3.3.4. Hash Maps for Dimensional Adaptivity

I must at this point admit, that I have not succeeded in finding publications on the topic of data structures for DASG. The assumption that many unpublished implementations are based on hash maps has, however, been confirmed in private communication. The same seems to be the case for implementations of the CT, which have similar requirements: The underlying task is to link each level vector to the memory location, where the associated subspace (or combi-grid) is typically stored in full grid layout. In Sect. 3.3.5, I discuss how this is best achieved for regular and truncated grids of a priori known shape. For DASG of unknown shape, a more flexible yet easy-to-manage solution is needed.

In [49], a data structure specifically developed for common CT functionality is described. Instead of a hash map, an array-based registry is maintained containing an indexed list of all level vectors in the current set. Relative indices stored for hierarchically dependent combi-grids are used for navigation inside the database, which, according to the authors, allows the efficient completion of all tasks related to grid management (for the CT). Fast and convenient lookups of arbitrary level vectors are, however, not directly
supported. Furthermore, the contents of the registry are not expected to be changed (merely extended), as altering one entry would entail an update of all other entries in the entire database.

In contrast, hash maps are highly flexible when it comes to finding arbitrary items or modifying the contents, which is what makes them attractive for SASG in the first place (cf. Sect. 3.3.2). Their major drawback is the lack of control one has over the managed memory and the layout of the stored data. But even though DASG only need a single lookup per subspace (whereas SASG perform lookups for every single grid point) the problem persists. After all, even only hundreds of processor cycles wasted in unnecessary lookups can hurt in an optimized implementation that could otherwise use these cycles to process whole subspaces. Experiments around the linear hierarchization algorithm have confirmed this suspected impact of hash map lookups. A program version relying on hash map lookups in the algorithmic phase has turned out to be several times slower (factor varying with the problem size) than a second version reading the subspace offsets sequentially from precomputed buffers. Moreover, precomputing and buffering complex traversals might also prove useful in DASG implementations customized for accelerators that do not support hash maps.

In order to not repeat in length the obvious from Sect. 3.3.2, I close this discussion of hash maps for DASG with a rather brief summary of possible types of keys. Note that this summary only reflects how I rate the general options, as this exact topic still seems to be unaddressed in publications.

1.) One can bet on the quality of distribution achieved through (raw) strings or randomly constructed integral type keys, which can both be supplemented with information about leaf subspaces.

2.) One can choose a representation supporting arithmetic directly on the key. The key type could be endowed also with other helpful information, for instance to make tasks related to grid management easier.

3.) Or one can make use of an enumeration scheme for subspaces and only store the obtained integer index.

### 3.3.5. Bijective Mappings

The last kind of data structure discussed here is also the most specialized one. It bases on the aforementioned mapping between the set of multi-dimensional sparse grid points and a dense index range. A linear array then suffices to compactly pack all grid coefficients. It is clear that such mappings can only exist for grids whose structure is a priori known. This is the case for regular and truncated sparse grids.

For such grids, I describe two bijective mappings that follow completely different philoso-
Co-Design, Part I: Formulating the Challenges

phies. Yet, both have in common that the full data structure merely consists of a small lookup table of size $d \times n$. I call these data structures \textit{implicit}, as no explicit information about the memory location of coefficients is kept. For random access, such information must thus be recomputed on the fly, which typically involves $O(d)$ or $O(d + n)$ steps (with small constant factor). Both implicit data structures avoid this cost by relying on efficient co-designed (parallel) iterators. Another advantage of such iterators is that they can directly be adapted to the particular data access patterns of algorithms.

Subspace-Based Approach

The first storage form takes advantage of a regular sparse grid’s composition of levels of equal-sized subspaces. The approach first appears in [37] and is further discussed in [88]. In my explanations, however, I stick to the slightly modified version published in [94] and extended to support truncated grids in [93]. In the latter publications, my colleagues and I demonstrate that porting our CPU implementation to GPUs is possible and beneficial. Note that all these storage forms (including the one sketched for DASG in Sect. 3.3.4) only differ from each other in the way the offsets of the subspaces are determined. The same 1-D operators can thus be reused for all these data structures without any modification.

For both sparse grid types, regular and truncated, composite mappings are defined that

1.) store grid coefficients according to increasing level,

2.) use combinatorial math to enumerate the set of level vectors on each level,

3.) map $d$-dimensional index vectors to the points of linearized $d$-dimensional full grids.

Step 2.) therein presents the non-trivial part. As detailed in [94], the problem can be reduced to integer partitioning, i.e., it is the same task as splitting a positive integer into a number of non-negative summands. The number of combinations for the set of level vectors on level $n$

$$\left\{ \vec{l} \in \mathbb{N}^d \mid |\vec{l}|_1 = n + d - 1 \right\}$$

is therefore given by the binomial coefficient

$$C_{d+n-2}^{d-1} := \frac{(d-1) + (n-1)}{d-1} = \frac{(d-1) + (n-1)}{n-1}. \quad (3.37)$$

Skipping some steps of the process at this point (see details in [94]), a zero-based enumeration for the level vectors can be derived for regular grids:

$$\text{subspaceidx}(\vec{l}) := \sum_{t=2}^{d} \left( \frac{t-1 + |\vec{l}_{0:t}|_1}{t-1} \right) - \left( \frac{t-1 + |\vec{l}_{0:t-1}|_1}{t-1} \right)$$

$$= \sum_{t=1}^{d-1} \left( \frac{|\vec{l}_{0:t+1}|_1 - 1}{t} \right) - \left( \frac{|\vec{l}_{0:t}|_1}{t} \right). \quad (3.38)$$
3.3. Commonly Used Sparse Grid Data Structures

Figure 3.10: The images highlight similarities between the pseudorecursive (left) and the subspace-based (right) enumeration schemes. A few points are marked consistently in both representations to make the comparison easier. Note that the axes in the subspace tableau on the right are oriented differently compared to other images in this work. Consecutively indexed points of a subspace thus appear horizontally aligned in both illustrations.

again with $\vec{r} = (\vec{l} - 1) \in \mathbb{N}^d$ and $\vec{r}_{j:k} = (r_j, \ldots, r_{k-1}) \in \mathbb{N}^{k-j}$. The full composite mapping $sb$ (subspace-based) (3.39) for regular grids takes $\mathcal{O}(d)$ time, and its correctness can be validated via the bold indices given in the subspace tableau in Fig. 3.10.

$$sb(\vec{l}, \vec{i}) := \sum_{\ell' = 1}^{\lceil l_1 \rceil - d} 2^{\ell' - 1} \cdot C_{d+b-2}^{d-1} \cdot \text{subspaceidx}(\vec{l}) + \sum_{t=0}^{d-1} 2^{i_t} \cdot \frac{i_t}{2}, \quad (3.39)$$

Expressions 1.) and 2.) in (3.39) depend on binomial coefficients, which can be cheaply precomputed and stored in a small lookup table. In the extension to truncated grids, this table is simply altered by deducting the contribution of the truncated parts (see [93] for the details). As suggested in [94], boundaries can be consistently integrated by adding a fourth level to $sb$: Boundaries of sparse grids are lower-dimensional sparse grids themselves, i.e., a regular sparse grid with boundaries can be represented as a sequence of smaller regular sparse grids, each of which is stored according to the mapping $sb$.

The decomposability and the extreme storage efficiency of the representation have led to excellent results for parallel implementations of hierarchization and evaluation [94, 93]. In [24], my colleagues and I enhance the approach specifically for the hierarchization algorithm. The core idea is to change the coefficient alignment within subspaces dynamically but in accordance with the respective algorithmic phase that is carried out. This allows the auto-vectorizer to enter the game. Since this feature is adopted in the data structure presented in Chapt. 4, the discussion of details takes place there.
Pseudorecursive Approach

The second bijective mapping is the bfs mapping \(3.31\) introduced in Sect. 3.3.3 in the context of the key-value approach for SASG. The mapping’s construction and some theoretical aspects are covered in that section, however, the matter of performance is not. The focus of the next few paragraphs is therefore on actual experiences made with this approach and about optimizations made possible by the absence of adaptivity. Figure 3.10 shows a visual comparison of this mapping with the subspace-based one.

It is intuitively clear that a priori knowledge about the grid’s structure can be used to perfect the grid traversal. But even though the traversal is precomputable also for this data structure, developing fitting algorithms is much less straightforward compared to the subspace-based approach of the preceding subsection. The storage layout – basically a “flattened” recursive grid traversal – enforces recursiveness in the algorithmic treatment, even though the used storage structure is a plain one-dimensional array. This is one of the reasons why I find it suitable to speak of a pseudorecursive approach.

In [78], the author explains the original intention pursued with the bfs mapping. Adopting ideas from the theory of I/O-efficient algorithms, he tries to localize operations by rearranging the coefficients in between computational phases. These efforts aim at the formulation of optimal algorithms, not only regarding runtime complexity, but also with respect to the I/O-model introduced in [3]. Proof of concept is achieved via the representative example of the hierarchization algorithm, which is very similar to the one sketched for the key-value approach. Only the sorting step becomes moot in the non-adaptive version, as coefficients can be reordered via a precomputable permutation. Besides improving general data locality, the permutation is also designed such that the same 1-D operator can be applied in all passes of the loop over the dimensions.

At the time of its invention, this approach to hierarchization beat all published implementations of regular grids. However, applying the idea of “moving data” to the subspace-based storage layout has led to even better results [24]. The reasons are more local data permutations and superior coefficient alignment for vectorization.

While apparently not optimal for hierarchization (and thus probably the unidirectional principle in general), the pseudorecursive data layout achieves so far unsurpassed performance for evaluation [25]. Its structure supports virtually all ideas for optimized sparse grid evaluation described in Sect. 3.2.2. The developed set of pseudorecursive algorithms outperforms actual recursive schemes and schemes employing the subspace-based storage form. These circumstances are reason enough to repeat once more the central conclusion of this section’s study of existing data structures: There is no ultimate data structure for all tasks, but some data structures are better suited for particular tasks than others.

---

\(^4\)The close collaboration with the author R. Jacob gave me the opportunity to access the algorithms and performance results long before the publication of [78] in 2013.
4. Co-Design, Part II: Presenting Solutions

In the first part of this solution chapter, I formulate a set of principles for efficient sparse grid implementations based on the insight gained in the analysis of algorithms and data structures in the previous Chapt. 3. The second part contains a detailed description of the main data structure for DASG that emerged from the development process guided by these principles. The focus is then on the core tasks again in a discussion with threefold interest:

1.) Does the new data structure enhance the formulation of the algorithms?
2.) Does the new data structure enhance the performance of the algorithms?
3.) Does a modified (or different) data structure serve the algorithm better?

Self-contained performance analyses for each task back up the choices made and put the presented solutions into perspective with competing implementations. The last part of the chapter contains an overview of the developed software components. Therein, the software’s modularity is demonstrated and explanations regarding extensibility are given in how-to style, targeting developers interested in a quick start with the code.


Despite its shortness, this section plays a key role in this thesis. A set of design principles is established, derived from the conclusions of the previous chapter’s analysis of typical data dependencies, data access patterns, and other well-proven software solutions. The formulation is kept general and to-the-point, omitting implementation-specific aspects that would hinder a broader applicability. For the remainder of this work, the principles serve as primary guidelines in the implementation of efficient DASG functionality. But this is not where their validity ends: The fact that their consistent application further have led to the probably fastest SASG implementation currently available is proof of that (for details see Sect. 5.4).
Use / switch to the right data structure: This is paramount if the cost of a program’s computational hot spot is to be lowered. The right data structure can be anything, even a (lossy) simplistic representation just carrying the relevant information. Of course, the cost of switching must always be considered in a cost-benefit analysis.

Use parallel iterators for efficient traversal and data access: Iterators can be designed to perfectly match the requirements of algorithms, especially regarding localized memory access. Only two different iterators are needed to cover the standard algorithms (unidirectional principle and evaluation). Iterators further signify a complexity reduction for users, since specifics of the data structure, the parallelization scheme, or even the platform can be hidden behind unified high-level interfaces.

Formulate 1-D operators as stencil-like kernels: In a DASG, a 1-D operator’s scope of operation is always a binary tree of a priori known shape. This motivates the formulation of operators as (potentially non-recursive) stencil-like kernels parameterized with the 1-D level. The use of temporary buffers during the stencil application may further enhance the approach: First, buffers localize repeated access to the tree data, and second, the stencil formulation can be purely 1-D, if the buffer’s interface hides everything about the currently processed dimension.

Vectorize stencil application across uniformly shaped trees: According to a sparse grid’s recursive composition of smaller sparse grids, any DASG of moderate size contains many 1-D trees of identical shape. If data alignment can be ensured without much effort, such uniform trees can be processed simultaneously in vectorized stencil code. The aforementioned buffers for the 1-D trees may again be of use here.

Minimize resource usage through in-place calculations: Above all, fast memory is a precious and scarce resource. This includes registers, caches, and even main memory in large scale computing. By minimizing the number of data copies, the load on caches and memory channels can be lowered considerably. Remember the unidirectional principle: One data copy is needed per level of recursion, however, this already assumes in-place processing.

4.2. A Proactive Data Structure for Dimensionally Adaptive Sparse Grids

The discussion of data structures in the previous chapter already covers a large variety of sophisticated solutions that have been developed and refined over a period of around 24 years. On the one hand, these prerequisites lower the chances of finding a completely different and superior approach. But on the other hand, it means that well-tried solutions with particular merits exist, which can provide good starting points for new endeavors.
Consequently, the data structure presented here avails itself of a well-studied basic construction that has been profitably employed in other implementations before. What makes it unique is the careful selection and beneficial consolidation of both established mechanisms and novel ideas. The process of assembly is carried out in full accordance with the principles laid down in Sect. 4.1 and the steps of this process are the topic of the following subsections. Afterwards, the data structure is thoroughly examined while it is used for solving the standard tasks. Compelling results prove the success of the concept.

### 4.2.1. Separation of Data and Structure: Ensuring a Versatile Design

There are several reasons why the separation of the actual data from the data structure makes sense, but above all a separate array for the coefficients is the most generic and versatile representation. It disconnects the data structure from the coefficient data type, plus, stripping the actual data from structural information generally increases locality and the chance of beneficial alignment (e.g., for vectorization or vectorized loading). Moreover, switching between (suitable) data structures becomes trivial, which can – as is shown – prove advantageous during computationally intensive phases of a program.

Just like for many other (unpublished) DASG data structures, a hash map forms the foundation of the data structure presented here. Its keys are basically the level vectors, and for each level vector an integer offset is stored, marking the corresponding subspace’s location in the global coefficient array. Within the array, subspaces are stored as linearized full grids, forming a sequence without gaps. In the following, I describe the distinct features that separate this particular solution from similar approaches. First, I mention how the specification of the key type can impact both algorithms and their performance. Next, I explain the subtleties and merits of the chosen coefficient layout.

**Hash Map and Key Type**

The main interest in these paragraphs is in the key type rather than the hash map, which is an off-the-shelf product. Figure 4.1a explains down to the bit how subspace-related information is encoded in 64-bit integers, and how the template parameter \( n \) of the key type’s C++ class `LevelVectorKey10nd<n>` controls the number of integers concatenated. Since up to ten components of a level vector fit into a single integer, an object of type `LevelVectorKey10nd<n>` supports up to \( d \leq 10 \cdot n \) dimensions. Note that a default value of \( n = 2 \) makes sense, as this choice does not lead to a measurable performance degradation even for small \( d \leq 10 \). This is because \( n \) is a compile-time constant, which lets the compiler remove all key-related loops from the code.

\footnote{It is the unordered map that is part of the STL since C++11.}
4. Co-Design, Part II: Presenting Solutions

(a) hash key type

(b) coefficient layout

Figure 4.1.: The basic components of the presented data structure are graphically explained. Figure 4.1a describes how arbitrary level vectors are encoded in (concatenations of) 64-bit integers. The code snippet at the top shows the declaration of the templated key class LevelVectorKey10nd<n>. It essentially consists of a 64-bit integer array of static length \( n \geq 1 \). However, a second view on the array elements is obtained via C++ unions, through which each 64-bit integer is seen as a composition of three parts: the dimensionality \( \text{dim} \) (4 bits), a collection of leaf flags \( \text{leaf} \) (10x 1 bit), and the level vector \( \text{vec} \) (10x 5 bits). This composition is illustrated at the bottom for LevelVectorKey10nd<1>. Preserving direct access to the key type’s integer member \( \text{key} \) is important for the efficient hashing and comparing of keys. Note also that limiting the representation of level vector components to 5 bits is synonymous with specifying an implicit truncation vector \( \vec{c} = (30, \ldots, 30) \) for every DASG (which is acceptable). This is because values 0 and 1 mark boundaries and are interpreted as \(-2\) and \(-1\), respectively, which leaves thirty more numbers to express the 1-D level. Figure 4.1b shows the subspaces of a DASG with \( \mathcal{L} = \{(3, 1), (2, 3)\} \) and boundary vector \( \vec{b} = (3, 2) \). The given subspace offsets comply with the constraints defined for the hash map: Level 3 subspace \((3, -1)\) is 32-byte-aligned at offset 0, which automatically aligns all subspaces of level \( l \geq 3 \) (white background) as demanded by AVX for four-way vectorization of 8-byte floats. Level 2 subspaces (yellow) are at least 16-byte aligned (sufficient for two-way vectorization via SSE), as they are all stored before the first level 1 coefficient (red).
A smart choice for the key type sometimes helps to circumvent costly hash map lookups. The LevelVectorKey10nd class is to this end supplemented with additional structural information, namely leaf flags for all $d$ dimensions. Let $\text{leaf}(k, j)$ be true if the leaf flag (1 bit) of key $k$ is set for dimension $j \in \mathcal{D}$. Let $\text{key}(\vec{l})$ further map level vector $\vec{l}$ onto a LevelVectorKey10nd instance with all leaf flags unset, while $\text{key}_L$ – depending on a DASG’s level vector set $\mathcal{L}$ – returns a key object such that

$$\forall \vec{l} \in \mathcal{L} : \text{leaf}(\text{key}_L(\vec{l}), j) \Leftrightarrow \vec{l} + \vec{e}^{(j)} \notin \mathcal{L},$$

where $\vec{e}^{(j)} \in \mathbb{N}^d$ is the $j$-th unit vector. A hash map can thus be formalized as

$$\mathcal{H} := \left\{ (\text{key}_L(\vec{l}), m) \mid \vec{l} \in \mathcal{L} \land m \in \mathbb{N} \text{ is offset of subspace } \vec{l} \right\}.$$  

Finally, let $h$ be the hashing function internally used by $\mathcal{H}$. Often, the hashed value is demanded to be an integral type. This renders the key member of the LevelVectorKey10nd class a perfect candidate. However, the leaf bits of the hash keys internally stored by the hash map can change (e.g., due to adaptive refinement), and random lookups will generally be unaware of whether a level vector $\vec{l}$ corresponds to a leaf subspace or not. Therefore, the leaf bits must be masked during hashing, such that

$$h(\text{key}(\vec{l})) = h(\text{key}_L(\vec{l})), \quad \forall \vec{l} \in \mathcal{L}$$

holds and $h(\text{key}_L(\vec{l}))$ can be found. How the leaf flags in fact help to lower the number of hash map lookups will be explained in the upcoming discussion of iterators.

**Coefficient Storage**

The view on a (non-adaptive) sparse grid’s approximation space as superposition of subspaces naturally encourages implementations that reflect this structure. And why not? A subspace’s coefficients fit nicely into a multi-dimensional full grid, whose serialization into linear storage is obvious. It is therefore not surprising that virtually all fast implementations of DASG use this efficient storage form – including implementations of regular and truncated grids, but excluding those based on the pseudorecursive approach.

Still, there are differences, especially in the handling of boundaries. Consider a non-overlapping basis such as the piecewise $d$-linear one. The solution presented here relies on the understanding of a subspace as collection of non-overlapping functions whose combined supports cover the entire domain. Boundary functions associated with opposite domain boundaries must then be assigned to different subspaces. Consider now the boundaries an extension of level 1 rather than a separate level 0. With this perspective, a consistent notion of subspace levels can be derived. First, the level vector notation is extended, and left boundaries are marked with $-2$ and right boundaries with $-1$. Next, the $L_1$-norm of a level vector $\vec{l}$ is redefined as

$$|\vec{l}|_1 := \sum_{j \in \mathcal{D}} \max \{l_j, 1\}.$$  


4. Co-Design, Part II: Presenting Solutions

The level \( n \) of subspace \( \vec{l} \) is then still given by \( n := |\vec{l}|_1 - d + 1 \). Two more advantages of this design are noteworthy:

- The confusion caused by the existence of a “boundary level” \( 0 \) is avoided in the multi-dimensional case, where even hierarchically dependent points on the domain’s faces would normally count to this single boundary level. Instead, the new notation leads to a clear assignment of all equal-sized subspaces to the same level.

- A boundary vector \( \vec{b} \in \{0, 1, 2, 3\}^d \) provides a straightforward way to specify for each dimension whether and where there should be boundaries. For this, a 2-bit code is used in the vector’s components: The less significant bit denotes the existence of the left boundary, the more significant bit stands for the right boundary (example: a left boundary exists for values 1 and 3).

Concerning the actual storage of subspaces, a small trick with big effect greatly favors the coefficient alignment for vectorized loading. As is graphically explained in Fig. 4.1b, the small offsets can be assigned to the large subspaces. Consider a 32-byte aligned coefficient array (as demanded by AVX), in which all subspaces on levels \( n \geq 3 \) have smaller offsets than the remaining subspaces on lower levels. In a setting where 8-byte doubles are used, all subspaces large enough for vectorized processing via AVX are then perfectly aligned at 32-byte boundaries. Caching benefits from the trick, too, as the aligned subspaces occupy a minimum number of cache lines (which generally count 64 bytes these days). This in turn reduces the risk of false sharing effects, which could otherwise occur whenever neighboring subspaces are concurrently processed by different cores (it can still happen for small unaligned subspaces).

4.2.2. Iterator-Based Traversal: Becoming Part of the Algorithm

The design principles demand the existence of iterators that correspond to the needs of algorithms. The good news is that only two iterators are needed to cover the algorithms of interest.\(^2\) I now look at the iterator for the unidirectional principle. Subsequently, I discuss the iterator for evaluation.

A Subspace Stack Iterator for the Unidirectional Principle

Regardless whether it is the linearized form found in transformations or the general form, the unidirectional principle postulates the application of 1-D operators to all 1-D trees along every dimension. A suitable iterator therefore delivers these 1-D trees as fast as possible and in a parallel fashion. Compared to an SASG, this is much more efficiently accomplished in subspace-based DASG representation, as hierarchically dependent subspaces already bundle many 1-D trees together. After all, it is by definition

\( ^2 \)The hash map usually provides an iterator over all subspaces (and thus indirectly over all points).
4.2. A Proactive Data Structure for Dimensionally Adaptive Sparse Grids

that a sparse grid’s subspaces optimally group points with equal dependency patterns: All points of a subspace originate from the same refinement process (which is documented in the components of the level vector).

A fitting iterator needs to visit all 1-D trees with respect to the processed dimension. One way to accomplish this is by searching the hash map for keys of root subspaces. However, these keys contain no information regarding the depth of the 1-D trees, which will inevitably lead to one bad lookup per tree. This is one scenario where the keys’ leaf flags come in handy. Seeking out the 1-D leaf subspaces instead of the roots immediately delivers the level of all 1-D hierarchies. Application of a 1-D operator in dimension \( j \in D \) is then achieved by

1.) iterating over all hash map elements \((k, m)\),
2.) identifying those \( k \) for which \( \text{leaf}(k, j) \) is true,
3.) constructing the keys of the hierarchical ancestors in dimension \( j \),
4.) looking up the offsets associated with the constructed keys,
5.) forwarding the offsets of this stack of subspaces to the operator’s algorithm.

Parallelization of these steps is straightforward, as the elements found in step 2.) can simply be dispatched to different worker threads. Another option is given by a round-robin-like static load balancing scheme, in which each of the \( p \) processors accepts only every \( p \)-th element that it finds. Either way, the parallelization can be realized via OpenMP, and it can be easily hidden from the user behind the iterator’s interface.

The iterator described above satisfies all previously stated requirements, only the expensive lookups in step 4.) can become a problem if ultimate performance is sought. But a very simplistic data structure already suffices to avoid these repeated lookups at operator application time: The traversal only needs to be carried out once during program initialization such that all offsets can be computed and logged in array buffers. Later, these buffers serve as proxies for the hash map and are streamed back efficiently to the single processors. Of course, the whole mechanism can be hidden behind the same iterator interface as used before.

An Iterator Supporting the Reuse of Partial Tensor Products during Evaluation

The analysis of evaluation in Sect. 3.2.2 underlines that the reuse of partial tensor products should be considered in efficient evaluation schemes. Obviously, the approach is most effective, if the chosen traversal scheme maximizes the number of reusable products. The 2-D variant of this scheme is illustrated in Fig. 3.3a. Its generalization for a \( d \)-dimensional grid with level vector set \( \mathcal{L} \) starts with the evaluation of the grid’s root subspace, and in every step it
1.) determines the index \( j := \arg\min_j \{ \bar{t} + \bar{e}(j) \in \mathcal{L} \} \) for the current subspace \( \bar{t} \in \mathcal{L} \),

2.) sets \( \bar{t} := (1, \ldots, 1, l_j + 1, l_{j+1}, \ldots, l_{d-1})^t \),

3.) computes all new partial products by reusing the partial product \( \prod_{k=j}^{d-1} \phi_{l_k, \bar{t}_k}(\bar{x}_k) \),

4.) looks up and evaluates the new subspace \( \bar{t} \).

Although the leaf information of key \( k = \text{key}_L(\bar{t}) \) already reduces the necessary hash map lookups per evaluated subspace to the one performed in step 4.), this single lookup is still too expensive. Fortunately, the new key type facilitates an elegant solution to this problem: Sorting the hash map’s keys in ascending order with respect to the class member vec (i.e., dim and leaf must be masked) yields the very sequence of subspaces that is also produced by the traversal scheme above. An efficient iterator can therefore copy all tuples in the hash map upon initialization, sort them according to the keys, and buffer the sorted list for fast access. Since evaluation is best parallelized across different evaluation points, a sequential implementation of such an iterator is all that it takes.

### 4.2.3. Task Awareness: Anticipating Data Access

The following paragraphs are about extensions to the basic data structure that facilitate the efficient and convenient treatment of problems requiring the unidirectional principle. Some of the ideas also find their way into the implementation of evaluation described in Sect. 4.3.2 but for now evaluation is excluded from the discussion.

The subspace stack iterator from the previous subsection presents a powerful tool. Yet, a gap still exists between the iterator that delivers the subspace offsets and the operator that works on 1-D trees, a gap that should be bridged from the data structure side. After all, the iterator already possesses perfect knowledge of the multi-dimensional subspace stack’s memory layout, while there is no reason why the programmer of a 1-D operator should be concerned with anything but the one-dimensional formulation. I conclude this section with the description of a solution that is well aligned with the formulated design principles. However, to understand the details, it is necessary to first dive into the specifics of the memory layout of subspaces.

### Data Alignment for Vectorization

Consider the 3-D subspace \( W_{(2,3,3)} \) and its hierarchical ancestors with respect to the \( y \)- and \( z \)-dimension drawn in Fig. [4.2]. Let the coefficients’ storage order within each 3-D subspace be \( xyz \), i.e., the index within a subspace’s full grid runs fastest along the

\[ \text{If the grid has boundaries in dimension } k < j, \ l_k \text{ must be reverted to } -2 \text{ (or } -1) \text{ instead of } 1. \]
4.2. A Proactive Data Structure for Dimensionally Adaptive Sparse Grids

Figure 4.2.: The 3-D subspace \( W_{(2,3,3)} \) is shown together with its hierarchical ancestors with respect to dimensions \( y \) and \( z \). All subspace coefficients reside in memory in \( xyz \) storage order as indicated for subspace \( W_{(2,3,2)} \) in the upper right corner. Green (blue) planes mark groups of grid points with the same coordinate in \( z \) (\( y \)) direction. The 1-D hierarchy at the bottom (on the left) symbolizes how one can abstract over dimensions \( x \) and \( y \) (\( z \) and \( x \)) when the work dimension is \( z \) (\( y \)).

\( x \)-direction and slowest along the \( z \)-direction (cf. red arrows in the upper right corner of Fig. 4.2). Let further be \( z \) the current work dimension, in which a 1-D operator is to be applied. In Fig. 4.2, the respective hierarchy of subspaces is aligned horizontally. Green planes in each subspace group those grid points that share the same \( z \)-coordinate. In the likenesses of the subspaces drawn below the hierarchy, the green planes are labeled with 1-D level-index pairs according to the planes’ \( z \)-coordinates \( (z = 0.5 \rightarrow (1,1), z = 0.25 \rightarrow (2,1), \) and so on). Because of the \( xyz \) storage order and the memory alignment of subspaces (cf. Sect. 4.2.1), the grouped grid points’ coefficients are perfectly aligned for vectorized processing. Given these circumstances, a strictly one-dimensional operator formulation can abstract over the other \((d-1)\) dimensions and process all 1-D trees inside the stack of subspaces at once.

In [24], a similar analysis of the coefficient layout leads to an efficient hierarchization scheme for regular grids. The length of aligned coefficient vectors within subspaces is
4. Co-Design, Part II: Presenting Solutions

therein maximized by ensuring that the work dimension’s index is always the storage order’s slowest index. This is achieved through cyclic dimensional shifts between the $d$ computational passes, resulting in the following sequence for a 3-D sparse grid:

\[
\begin{align*}
\text{xyz order} \xrightarrow{\text{shift}} & \text{yzx order} \xrightarrow{\text{shift}} \text{zxy order} \xrightarrow{\text{shift}} \text{xyz order} \\
\end{align*}
\]

The main disadvantage of the approach is that the dimensional shift of a subspace generally requires a permutation of all its coefficients in extra memory. Furthermore, it is inconvenient to keep track of the current data layout during the execution of the general $UpDown$ scheme, where the order of the processed dimensions is not always fixed. Close inspection of the subspaces’ storage layout will show now that effective vectorization can be achieved with much less effort.

In the following, the work dimension is fixed to $k \in D$, and all $d$-D subspaces are stored in generalized $xyz$ storage order. Let $W^{(DP)} = 2^w, w \in \mathbb{N}$ be the width of the platform’s double precision vector registers. For each level vector $\vec{l}$, consider the helpful vector $\vec{r} \in \mathbb{N}^d$, which is split into three parts such that

\[
\vec{r} = (r_0, \ldots, r_{k-1}, r_k, r_{k+1}, \ldots, r_{d-1}), \quad r_j = \max\{l_j - 1, 0\}. \quad (4.5)
\]

Subspace $\vec{l}$ has $2^{r_k}$ coefficients, which can be aligned in $2^{r_k}$ vectors of length $C \cdot S = 2^{r_k} - r_k$ for vectorized processing. These vectors are related to the 1-D hierarchy with respect to dimension $k$, let them therefore be denoted by

\[
(l_k, i_k)_0:CS, \quad i_k \in \mathcal{I}_k. \quad (4.6)
\]

An abstract view of the subspace’s storage layout (left to right, top to bottom) is then given by

\[
\begin{align*}
(l_k, 1)_0:S, & \quad (l_k, 3)_0:S, \ldots, (l_k, 2^{r_k+1} - 1)_0:S, \\
(l_k, 1)_2:S, & \quad (l_k, 3)_2:S, \ldots, (l_k, 2^{r_k+1} - 1)_2:S, \\
\vdots & \quad \vdots, \quad \vdots, \quad \vdots, \quad \vdots, \\
(l_k, 1)_{(C-1):CS}, & \quad (l_k, 3)_{(C-1):CS}, \ldots, (l_k, 2^{r_k+1} - 1)_{(C-1):CS}.
\end{align*}
\]

(4.7)

Obviously, two extreme cases exist: The best case is $k = d - 1$, because the chunk size $S$ is maximized ($C \cdot S = 1 \cdot S$). The worst case is $k = 0$, because the chunk size is at its minimum $S = 1$, ruling out vectorized processing of the chunks. Generally, two conditions need to be checked,

\[
\begin{align*}
C \cdot S & \geq W^{(DP)} \\
S & \geq W^{(DP)},
\end{align*}
\]

(4.8)

(4.9)

\footnote{It holds $W^{(DP)}_{AVX} = 4$ on AVX supported platforms, $W^{(DP)}_{SSE} = 2$ on SSE supported platforms, and $W^{(DP)}_{AVX-512} = 8$ on Intel MIC.}
4.2. A Proactive Data Structure for Dimensionally Adaptive Sparse Grids

Figure 4.3.: The localized coefficient permutations are shown for the 3-D subspace $W(2,3,3)$. They enable 4-way vectorized processing with respect to dimension $z$ (left), $y$ (center), and $x$ (right). The illustration uses the grid points’ index vectors for identification of coefficients within the subspace. In all three coefficient matrices, the array index of the linearized full grid advances from left to right, top to bottom. The gray blocks in the background indicate the scope of the local permutations.

In order to decide whether vectorized treatment of subspace $\vec{\ell}$ is possible at all (cf. condition (4.8)), and whether the existing chunks are already sufficiently large for vectorization (cf. condition (4.9)). Figure 4.3 illustrates these relations at the example of the 3-D subspace $W(2,3,3)$. For work in the $z$-dimension ($k = 2$, left), no permutation is required as (4.9) is satisfied. The same holds for the $y$-dimension ($k = 1$, center), if SSE with $W(SSE) = 2$ is used. For AVX, however, several localized permutations – each involving $W(AVX) \cdot 2^{r_1} = 16$ coefficients – are required to produce chunks of size $W(AVX) = 4$. For the worst case $k = 0$ (right) (4.9) is unfulfilled, but (4.8) holds. Chunks of size $W(DP)$ can therefore be produced by permuting sets of $W(DP) \cdot 2^{r_0} = 8$ in the example).

Summing up, the dimensional shift that guarantees vectors of maximum length for work dimension $k$ involves transposing a rectangular matrix of size $2^{r_k} \times 2^{j_{k-1}-r_k}$. For this task, sophisticated approaches for vectorized in-place transpositions of arbitrary rectangular matrices exist (e.g., see [36]). These approaches typically identify small dependent subsets of matrix entries and permute them in constant size extra memory. However,
the matrices considered here are comparably small and are perfectly aligned due to
dimensions that are powers of two. For these reasons, straightforward transposition in
extra memory leads to better results. Nevertheless, one has to consider the absolute
cost of the transposition step, and this cost even tends to outweigh the cost of applying
a 1-D operator. The approach described above is superior (see results in Sect. 4.3.1),
as it localizes permutations and makes perfect use of pre-aligned data chunks. Permu-
tations become therefore moot in most cases, especially if the work dimension index $k$
approaches $(d-1)$ rather than 0. For the other cases, the localized permutations can be
realized using efficient shuffle instructions for the reordering of vector register contents
(SSE, AVX, AVX2, and AVX-512 all provide such instructions).

A 1-D Stencil View for Compute Kernels

The discussion in this section so far involved a hash map holding subspace offsets,
the hash map’s special key type, as well as co-designed iterators that are aware of data
dependencies and data locality in algorithms. Furthermore, the data layout in subspaces
allows for vectorized application of 1-D operators. All that is missing is a consistent
interface for the programming of such operators as demanded by the design principles.

Listing 4.1: An excerpt of the interface of C++ class SubspaceStackBuffer is shown.
The class bridges the gap between the subspace stack iterator from Sect. 4.2.2 and the algorithm in form of 1-D operator formulations.

class SubspaceStackBuffer {
public:
  :
  /* DATA PREPARATION */
  // derive 1-D level n, chunk size S, and chunk count C
  void setupBuffer(int workDim, const int* leafVector);
  
  // apply permutation to "data" if necessary
  void loadSubspace(int level1d, double* data);
  // revert permutation of "data" if previously applied
  void writeBackSubspace(int level1d, double* data);

  /* DATA MANIPULATION */
  // set components of vector $(l,i)_0:CS$ to a
  void set(int l, int i, double a);
  // $(lx,ix)_0:CS += a*(ly,iy)_0:CS$
  void xpay(int l(x), int i(x), double a, int l(y), int i(y));
  
};
Listing 4.1 shows parts of the declaration of class SubspaceStackBuffer. The class addresses the remaining design principles laid down for the data structure in Sect. 4.1, as it

- uses the knowledge about the 1-D hierarchy’s shape to restrict the scope of operations,
- aligns coefficients automatically for vectorized processing,
- caches coefficients between calls to loadSubspace and writeBackSubspace,
- allows the manipulation of hierarchical coefficients via a 1-D interface (e.g., xpay),
- does with a slim and portable interface that hides platform specifics.

The class’s interface for data manipulation can be arbitrarily extended by obvious functions such as axpy or copy, which are all linear operations of the same kind. Non-linear combination of hierarchical coefficients does usually not occur, as a sparse grid interpolant is a linear combination of basis functions itself. As a consequence, it is rather straightforward to port these interface functions when another platform is targeted. Porting the hidden permutation kernels requires more thought, but the SIMT features of data parallel languages such as OpenCL can actually facilitate this task.

The so far unmentioned class Operator1d completes the picture, as users can here implement operator logic in form of a virtual function. The function takes a filled SubspaceStackBuffer as argument and allows a shader-like formulation of the 1-D operator: The abstracted coefficients of the 1-D hierarchy represent the primitives that can be manipulated through the linear operations defined by the buffer’s interface. Listing 4.2 concludes this section by bringing together all the presented components in an example that demonstrates their interaction. In only a few lines of code, a given operator is applied to a given grid along work dimension $k$. SubspaceStackIterator iter identifies those subspaces in the grid that form hierarchies with respect to the work dimension. These subspaces are loaded from array $u$ into SubspaceStackBuffer buffer, where they are cached and aligned for the upcoming application of one or more 1-D operators. For reasons of efficiency and correctness, this step is carried out by the static method apply of the Operator1d class. Note finally that iterator iter also offers a parallel mode that supports the concurrent application of operators to the contents of several subspace stack buffers in different threads.

---

5The names and signatures of some functions in the example do not fully match the actual source code. The changes are, however, only applied to keep unnecessary complexity out of the example. They do not alter the presented concept at all.
4. Co-Design, Part II: Presenting Solutions

Listing 4.2: The interaction of the presented data structure’s components is shown in the application of \textit{numOps} 1-D operators \textit{ops} to \textit{grid} along work dimension \(k\). The iterator \textit{iter} locates dependent coefficients, which are then loaded to \textit{buffer} in order to be prepared for efficient processing. The coefficients are manipulated through \textit{buffer}’s interface and are afterwards written back to the coefficient array \(u\).

\begin{verbatim}
void applyOperator1d(int k, Grid& grid, double* u, 
                    Operator1d* ops, int numOps)
{
    // create a buffer object and a grid iterator
    SubspaceStackBuffer buffer(grid.getMaxLevel());
    SubspaceStackIterator iter(grid);

    // iterate over all 1-D subspace hierarchies
    for (iter.begin(k); !iter.isEnd(); ++iter)
    {
        // prepare the buffer’s internal data layout
        buffer.setupBuffer(k, iter.getLeafLevelVector());

        // load subspace stack into buffer
        for (int l = 1; l <= iter.getLevel1d(); ++l)
        {
            size_t offset = iter.getSubspaceOffset(l);
            buffer.loadSubspace(l, u + offset);
        }

        // apply (possibly more than one) operator
        Operator1d::apply(buffer, ops, numOps);

        // write back updated coefficients
        for (int l = 1; l <= iter.getLevel1d(); ++l)
        {
            size_t offset = iter.getSubspaceOffset(l);
            buffer.writeBackSubspace(l, u + offset);
        }
    }
}
\end{verbatim}
4.3. The Core Tasks Solved

Complementary to the analysis of algorithms and data access patterns in Sect. 3.2, this section focuses on implementational aspects related to the solution of the core tasks using the new data structure(s). This includes many novel ideas concerning algorithmic optimization and in case of the evaluation task even a completely new data structure. The contents of this part are therefore no mere recapitulation of the approaches discussed in Sect. 3.2, they rather complete the picture, covering also aspects previously not addressed. The same holds for the examples used to illustrate how particular features of the presented solutions are best exploited for a given problem. For instance, the object of study in the discussion of basis transformations is the quite sophisticated prewavelet transform, for which a highly efficient 1-D operator is derived. The central questions asked for the unidirectional principle are about the availability and the prerequisites of algorithmic optimizations for the application of FE operators. Each thread of discussion is concluded with the presentation of performance results either from specifically tailored experiments or, where available, comparisons with other software. The excellent outcome of the experiments is proof of a spot-on analysis in Chapt. 3 and justifies the decisions and choices made in the course of implementation.

4.3.1. Basis Transformations

Section 3.2.1 already contains detailed discussions of all basis transforms, and for linear and polynomial hierarchization even the actual algorithms are given (cf. Alg. 1). Since these algorithmic tasks require no further elaboration, the first part of this section is on the more challenging prewavelet transform that involves the solution of several tridiagonal SLEs. I will present the C++ source code of a very concise operator formulation for this transform, and it will become clear instantly how a one-to-one mapping of the statements in Alg. 1 leads to valid hierarchization operators for the other bases as well. This is informal proof of the presented concept’s applicability, and it again underlines the merits of the presented data structure that automatically translates the user’s 1-D stencil code into a highly efficient vectorized (and parallelized) operator implementation. The second part of this section contains an extensive performance analysis, in which the new data structure competes against other implementations that are known to be fast.

The Prewavelet Transform

On level $l$, the prewavelet coefficients $\gamma_{l,i}$ are determined from the linear surpluses $\alpha_{l,i}$ according to (3.14), i.e., a tridiagonal SLE with system matrix $A \in \mathbb{R}^{k \times k}, k = 2^{l-1}$ of

---

*The C++ version of linear (de-)hierarchization is also included in Appx. A.1 for convenience.*
the form
\[
\begin{pmatrix}
1.2 & 0.4 \\
0.4 & 1.6 & 0.4 \\
& \ddots & \ddots & \ddots \\
0.4 & 1.6 & 0.4 \\
0.4 & 1.2
\end{pmatrix}
\begin{pmatrix}
\gamma_{l,1} \\
\gamma_{l,3} \\
\vdots \\
\gamma_{l,2l-3} \\
\gamma_{l,2l-1}
\end{pmatrix}
= 
\begin{pmatrix}
\alpha_{l,1} - t_{l+1,2} + t_{l+1,4}/2 \\
\alpha_{l,3} - t_{l+1,6} + t_{l+1,6\pm 2}/2 \\
\vdots \\
\alpha_{l,2l-3} - t_{l+1,2(l-3)} + t_{l+1,2(l-3\pm 1)}/2 \\
\alpha_{l,2l-1} - t_{l+1,2(l-1)} + t_{l+1,2(l-2)}/2
\end{pmatrix}
\]  

needs to be solved. The coefficients of \( A \) are constant, which allows to precompute the LU decomposition of \( A \) with matrices \( L \) and \( U \) as follows:

\[
A = \begin{pmatrix}
1 & d_1 & 1 \\
& \ddots & \ddots & \ddots \\
& & d_{k-2} & 1 \\
& & & d_{k-1} \\
& & & & 1
\end{pmatrix}
\begin{pmatrix}
e_{0} & 0.4 \\
1 & e_1 & 0.4 \\
& \ddots & \ddots & \ddots \\
& & 1 & e_{k-2} & 0.4 \\
& & & & 1/ar{e}_{k-1}
\end{pmatrix}
\]

At program runtime, the SLE is most efficiently solved if any floating point division is avoided in the inner loops. It therefore makes sense to directly precompute the reciprocal values \( e_i, 0 \leq i < k-1 \) of the actual diagonal elements of matrix \( U \), such that the forward and backward substitutions of the LU decomposition depend on the fast operations addition, subtraction, and multiplication only. The \( d_i \) and \( e_i \) for \( 1 \leq i < k \) can be determined according to the leapfrog scheme

\[
e_0 := 1.0/1.2, \\
d_i := 0.4 \cdot e_{i-1}, \\
e_i := \frac{1}{1.6 - 0.4 \cdot d_i},
\]

with \( 1 \leq i < k-1 \). The last element \( \bar{e}_{k-1} \) on \( U \)’s diagonal is determined via

\[
\bar{e}_{k-1} := \frac{1}{1.2 - 0.4 \cdot d_{k-1}}.
\]

The advantage of this element’s separate treatment lies in a reduction of the storage cost: The \( e_i, 0 \leq i < 2^l-1 \) on level \( l \) can be directly reused for the decomposition on level \( l' > l \), as the prefix of the \( e_i \)–sequence is equal on all levels. Only the formula for the last element of the matrices’ diagonals deviates from the general scheme. It thus suffices to store the \( e_i \) (and the \( d_i \)) for the maximum level \( n \), plus one additional value \( \bar{e}_{2^l-1} \) for each level \( 1 \leq l \leq n \). Altogether, \( 2^n - 2 + n \) coefficients need to be precomputed and stored for the \( d_i, e_i, \) and \( \bar{e}_i \).
Listing 4.3: The bottom up implementation of the prewavelet transform is shown. The algorithm assumes a generating system, therefore, a special implementation of the SubspaceStackBuffer interface is required that gives access also to the coefficients with even indices on each level. The tridiagonal SLE’s right-hand side is assembled in these coefficients on each level, before the SLE is solved in linear time via LU decomposition.

```c
void forward(int n, SubspaceStackBuffer* buffer) {
  // bottom up traversal of the hierarchy
  for (int ℓ = n; ℓ > 1; --ℓ) {
    int i_max = 2^ℓ - 1;
    // 1.) build right-hand side & resolve redundancy
    if (ℓ < n) { // no temporary values on highest level
      buffer->xpay(ℓ, 1, -1.0, ℓ+1, 2);
      buffer->xpay(ℓ, 1, 0.5, ℓ+1, 4);
      for (int i = 3; i < i_max; i += 2) {
        buffer->xpay(ℓ, i, 0.5, ℓ+1, 2i-2);
        buffer->xpay(ℓ, i, -1.0, ℓ+1, 2i);
        buffer->xpay(ℓ, i, 0.5, ℓ+1, 2i+2);
      }
      buffer->xpay(ℓ, i_max, 0.5, ℓ+1, 2i_max-2);
      buffer->xpay(ℓ, i_max, -1.0, ℓ+1, 2i_max);
    }
    // 2.) LU decomposition: forward substitution
    for (int i = 3; i <= i_max; i += 2) {
      buffer->xpay(ℓ, i, -d_i/2, ℓ, i-2);
    }
    // 3.) LU decomposition: backward substitution
    buffer->scale(ℓ, i_max, e_i_max/2);
    for (int i = i_max-2; i > 0; i -= 2) {
      buffer->xpay(ℓ, i, -0.4, ℓ, i+2);
      buffer->scale(ℓ, i, e_i/2);
    }
    // 4.) temporary values for next level
    for (int i = 2; i < i_max; i += 2) {
      buffer->set(ℓ, i, ℓ+1, 2i);
      buffer->xpay(ℓ, i, -0.6, ℓ, i-1);
      buffer->xpay(ℓ, i, -0.6, ℓ, i+1);
    }
  }
  if (n > 1) { // resolve redundancy on level 1
    buffer->xpay(1, 1, -1.0, 2, 2);
  }
}
```
This completes the list of ingredients for the realization of a linear runtime operator. Listing 4.3 shows its implementation as virtual function forward of the previously introduced abstract C++ class Operator1d. Despite the complexity of the task, the operator’s source code fits on a single page, which is a strong indicator for an intuitive programming interface with high usability. Note once more, that the programmer’s responsibility ends with the definition of rules for the hierarchy of coefficients in 1-D. The data structure then takes over and applies these rules correctly and, if desired, concurrently to all coefficients in the sparse grid.

For completeness, the inverse of the prewavelet transform is included as well in Appx. A.2. It is defined in Lst. A.2 as virtual function backward, forming the counterpart to function forward from Lst. 4.3. Together, these complementary virtual functions define hooks in the Operator1d interface that allow for side-by-side implementations of either transforms and their inverses, or the Up and Down operators of bilinear forms that are subject to an UpDown splitting. Although function backward comprises about the same number of statements and numerical operations as forward, the algorithms of both transforms differ greatly. When compared to the hierarchization schemes used for the linear and polynomial bases, the most noteworthy aspect is probably that Up operators are used for forward and backward. Both hierarchization schemes rely on Down operators, and the inverses are obtained by simply reversing the order and the “direction” of the operators’ numerical operations (which still leads to Down operators).

Performance Results

Now, the implementations of the transforms are put to the test. Benchmarks are only done on the dual-socket eight-core CPU system in order to stick close to the boundary conditions of potential target settings. Furthermore, only double precision floating point numbers are considered, as they are essential in virtually all applications related to classification or simulation. Due to the unavailability of competitive DASG implementations, codes for regular sparse grids are used in the comparisons. Of course, the DASG data structure is thus only challenged while storing regular grids sparse grids. The results obtained are nevertheless representative, as a reasonable degree of grid refinement is not expected to have an impact on the data structure’s overall performance. Another advantage of working with regular grids is that their structure is fully defined by just two parameters, the dimensionality and the grid level. In contrast, adaptively refined grids can assume arbitrary shapes, which is difficult to quantify in an analysis.

In the first test, the interest is in the strong scaling behavior exhibited by all three transforms. In Fig. 4.4 the speedups for up to 16 threads are compared for different combinations of dimensionality \(d\) and grid level \(l\). In the left column \(d\) and \(l\) are both

---

\(^7\)For consistency with the explanation of the LU decomposition, the coefficients \(d_i, e_i,\) and \(\bar{e}_i\) are included in the source code as originally defined. In a real C++ program, they would obviously be stored in linear arrays.
4.3. The Core Tasks Solved

varied, in the right column only $l$ is varied. Linear hierarchization (top) has an extremely low operational intensity and scales worse in the number of threads than the other transforms. At most two coefficient updates are done for every two memory operations, which is not enough to hide memory latency behind computation. For configurations with $T \leq 8$ threads on a single NUMA node, this results in mediocre parallel efficiency. For $T > 8$, some threads need to access the remote NUMA node’s memory, which causes a severe performance drop especially for grids with mostly small subspaces (see $l = 7, 9$). With growing subspace size (see $l = 11, 13$), the use of the second socket leads to a performance gain, as the increased memory bandwidth somewhat compensates for the latency effects.

Similar behavior is observed for the polynomial (center) and the prewavelet basis (bottom). However, the higher computational intensity generally results in better parallel efficiency. Most computational work needs to be done for the prewavelets, and so speedups of 6x and higher are achieved for $T \leq 8$ for all grids with level $l \geq 9$. For the largest examined grid size ($d = 13, l = 13$), this behavior even continues on the second socket, resulting in a speedup of more than 12x for $T = 16$.

Figure 4.5 examines the performance of the SubspaceStackIterator class. To this end, linear hierarchization is compared for the DASG data structure and an implementation of regular sparse grids relying on the bijective mapping from [94] (cf. Sect. 3.3.5). Subspace processing itself is identical in both programs, but different methods are used to determine the subspaces’ memory locations. Memory-wise, the SubspaceStackIterator class is more wasteful. It precomputes and buffers all memory addresses in the program’s offline phase and thus avoids costly hash map lookups in the online phase. The bijective map variant merely requires a small lookup table but needs $O(d)$ operations to calculate each subspace’s address on the fly. Except from the configuration with only one thread, all results are in favor of the DASG data structure and a speedup of nearly 8x is seen for some grid sizes. This result proves that efficient parallel grid traversal is at least as important as a set of highly optimized compute kernels. For the grid with boundaries, the speedups appear to be even slightly higher (cf. Fig. 4.5b) than for the grids without boundaries (cf. Fig. 4.5a). This is as expected, since managing boundaries in form of projected lower-dimensional grids makes the composite bijective mapping considerably more complex and raises the number of required lookup tables from 1 to up to $2^{d-1}$.

The previously fastest hierarchization algorithm published in [24] can be considered the predecessor of the solution presented here. Being the author of both codes allows me to carry out a component-wise dissection and comparison. As for the bijective mapping from [94] (which is also employed in [24]), the results in Fig. 3.3.5 prove that the new approach leads to better results in parallel settings. Now, the interest is only in the vectorization schemes. Figure 4.6 shows the speedups of the DASG data structure with its localized permutations over the dimensional shift approach from [24] (cf. discussion in Sect. 4.2.3). For larger grids, the new approach achieves speedups of up to 3x. For grids with smaller subspaces on the other hand, the entire dimensional shift is likely to be
computed efficiently in fast cache. Consequently, the shift’s low overhead is outweighed by the gain of coefficient vectors with maximum length, leading to results that beat the new approach by up to 31%.

To conclude the comparison of the new solutions with the previously fastest implementation of linear hierarchization, two things need to be pointed out:

- The overall speedup of the new approach is a combination of the speedups in Fig. 4.5 and Fig. 4.6. Thus, the new approach wins in almost all disciplines (except where the sequential treatment of small grids is concerned).

- The implementation benchmarked in [24] neither uses explicit vectorization, nor does it ensure optimal data alignment. However, all new experiments related to the dimensional shift account for both optimizations. This is likely to improve the position of the dimensional shift in the comparison, and yet, the new approach’s superiority is obvious.
4.3. The Core Tasks Solved

Figure 4.4.: Strong scaling results on the SNB-EP test platform are shown for the basis transformations from the nodal point basis to the linear (top), polynomial (center), and prewavelet (bottom) bases. Left column: The dimensionality $d$ and the level $n$ are both varied. Right column: Only the level $n$ is increased. The NUMA system’s second socket is only used for 9 and more threads.
4. Co-Design, Part II: Presenting Solutions

![Graph](image)

(a) without boundaries

(b) with boundaries

Figure 4.5.: The efficiency of subspace access during linear hierarchization of regular sparse grids is compared for the presented DASG data structure and one that computes the subspace offsets via a bijective map (cf. Sect. 3.3.5). The plots show the speedups achieved by the DASG storage form, if both implementations only differ in the computation of the offsets. Results for $T \in \{1, 4, 8, 16\}$ threads on the SNB-EP platform are given for grids of different size without (left) and with boundaries (right).

![Graph](image)

Figure 4.6.: The efficiency of the presented data structure’s vectorization scheme is compared to the dimensional shift approach from [24] (cf. Sect. 4.2.3) for the task of linear hierarchization. The plot shows the speedups obtained for the new vectorization scheme, when both implementations use identical hierarchization algorithms and iterators. The results are measured on the SNB-EP platform for regular grids of different size (without boundaries) and different numbers of threads $T \in \{1, 4, 8, 16\}$. 
4.3.2. Evaluation

The focus is now on settings where multiple grid evaluations need to be done as fast as possible. This task will be referred to as *multi-eval* from now on. Consider the semi-naïve subspace-based approach that processes one subspace after another (the naïve approach being the one described by (3.15)). The approach is straightforward and leads to good results on heterogeneous platforms [94, 92]. It even tends to benefit from data locality, as affected coefficients of different evaluations are likely to share cache lines, especially in the smaller subspaces. Nevertheless, the experiments on CPUs and GPUs in [25] clearly show the superiority of the pseudorecursive approach (see explanations in Sect. 3.3.5). Why this is, and what can be learned from these findings is shortly summarized in the following. The conclusions then lead to a new flexible approach for DASG, which is called for, as the scheme from [25] only covers regular and truncated grids.

A Flexible Hybrid Data Layout for DASG

The results in [25] underline that the pseudorecursive approach successfully combines those features that are most desirable in sparse grid evaluation. Not only does the data layout inherently favor the incremental computation of the tensor product, measurements also indicate a considerably lower number of L1 and L2 cache misses compared to the naïve subspace-wise evaluation. As a result, the approach leads to the by far fastest implementation of sparse grid evaluation currently published. On the downside, the pseudorecursive schemes tend to require extremely sophisticated algorithms that put high pressure on the scarce resources of GPUs (cf. [25]). Furthermore, its extension to generalized DASG is prevented by the data layout’s intrinsic structure. Another drawback is the global permutation of the coefficient array that is needed if applications switch from the DASG data structure’s subspace-based representation to the pseudorecursive layout. Although this operation is parallelizable, it involves scattered memory access across the full array and does therefore not scale well in the number of threads. To summarize these observations, the perfect approach would manage to combine the strengths of the pseudorecursive approach with the flexibility of a subspace-wise treatment.

The starting point is the flexible but “slow” subspace-based scheme. The good news is that one of the scheme’s major flaws can be instantly fixed by simply using the iterator described in Sect. 4.2.2. This is because the iterator visits the subspaces in the order that is optimal for the incremental evaluation of the tensor product. Additional support from the hardware prefetcher can be secured by actually arranging the subspaces in the order dictated by the iterator. From the memory perspective, a pass of multi-eval then

---

8 Since the order in which affected coefficients are visited by the pseudorecursive scheme is also optimal for the tensor product calculations, both orders must be identical. For a single grid evaluation and non-overlapping basis functions this is indeed the case. An understanding of these relations is best developed by considering the pseudorecursive data layout a sequence of subspace fragments. Use Fig. 4.9 to follow the trace of an evaluation in both layouts and verify this statement.
4. Co-Design, Part II: Presenting Solutions

Figure 4.7.: Left: The 1-D bfs enumeration is shown again for a 1-D tree of 3 levels. Right: All tree nodes contain the (zero-based) index at which they are stored on their respective level (from left to right). The highlighted path to node \((3,5)\) (bfs index 5) and its ancestors (bfs indices 2 and 0) is fully encoded by the value \textit{path1d}='10'. The node itself is stored on level 3 at index '10'=2, and its ancestors’ indices on their levels are obtained through bitwise right shifts of '10' (cf. '1'=1 and '0'=0).

The clustering of affected coefficients in 1-D bfs trees and these trees’ efficient processing corresponds to a sequential scan of the coefficient array.

The pseudorecursive approach also excels in terms of cache efficiency. The explanation is found in the sparse grid’s interpretation as recursive composition of 1-D subgrids. Each 1-D subgrid clusters several affected coefficients inside a compact 1-D bfs tree representation (cf. bfs mapping \((3.30)\)). What is more, evaluating the grid at point \(x\) means evaluating all of its 1-D subgrids at coordinate \(x_0\). This step can be abstracted as application of a precomputed 1-D evaluation stencil as shown in Alg.\,\[3\] The advantages are manifold: The formulation does without conditional branching, it is independent of the basis functions’ complexity, and it is equally applicable to grids with boundaries. As already detailed in \[25\], GPU implementations cannot afford to store such stencils in fast memory. And yet, they can still draw benefit from the uniform shape of the 1-D bfs trees as can be seen in Alg.\,\[4\] The adjusted variant precomputes an integer \textit{path1d} that encodes the descent in the bfs tree according to Fig.\,\[1.7\] The algorithm thus avoids conditionals and can efficiently derive the affected coefficients’ offsets, but basis functions must still be evaluated online.

**Algorithm 3** The algorithm applies a precomputed 1-D evaluation stencil \(s\) to the 1-D bfs tree of level \(n\) stored at position \textit{offset} in coefficient array \(\vec{\alpha}\). As can be seen, the evaluation coordinate \(x\) vanishes from the formulation, and with it, the need for conditionals vanishes as well.

\begin{verbatim}
1: function EVAL1DGGENERAL(s, n, offset, \vec{\alpha})
2:     result ← 0
3:     for \(l = 1,\ldots,n\) do
4:         result ← result + \(s.\phi[l] \ast \vec{\alpha}[\text{offset} + s.\text{offset}[l]]\)
5:     end for
6:     return result
7: end function
\end{verbatim}

The clustering of affected coefficients in 1-D bfs trees and these trees’ efficient processing...
4.3. The Core Tasks Solved

Algorithm 4 The GPU version of Alg. 3 uses a precomputed variable \textit{path1d} (cf. Fig. 4.7) to avoid conditionals in the evaluation of a 1-D bfs tree. However, shortage of fast memory disallows the storing of a priori evaluated basis functions. Note that \textit{path1d} is assumed to be given with respect to level \( n \).

1: function \textsc{eval1dGpu}(x, path1d, n, offset, \( \vec{\alpha} \))
2: result ← 0
3: for \( l = 1, \ldots, n \) do
4: \( \text{idx} ← (2^{l-1} - 1) + (\text{path1d} >> (n - l)) \)
5: \( i ← 2 \ast (\text{path1d} >> (n - l)) - 1 \)
6: \( \phi ← \max(0, 1 - |2^l \ast x - i|) \)
7: \( \text{result} ← \text{result} + \phi \ast \alpha[\text{offset} + \text{idx}] \)
8: end for
9: return result
10: end function

via precomputed stencils are definitely desirable in an implementation. Again, there is a very natural fix that adds these lacking features to the subspace-based evaluation scheme. The illustration in Fig. 4.8a demonstrates how bfs trees are easily produced by interleaving subspaces with hierarchical dependence in the first dimension. The bfs representation of the resulting spaces \( \tilde{W} \) permits the application of all optimizations just discussed, but the list of advantages continues:

- The representation is flexible enough to handle all DASG, including those with exotic boundary configurations (one-sided, two-sided, trapezoidal, etc.).
- The data layout sticks close to a subspace-wise coefficient grouping, permitting a faster and more scalable initial permutation than the pseudorecursive layout.
- The traversal is very natural and suitable even for “challenging” platforms such as GPUs.

The latter point is underlined by the sketched grid traversal in Fig. 4.8b. All information needed for navigating to the next \( \tilde{W} \) in the sequence can be encoded in just two numbers:

1.) a number \( k \) specifying which dimension is refined next
2.) the level \( n \) of the bfs trees that make up the next \( \tilde{W} \)

Everything else can be derived from the grid traversal’s context, which is typically expressed as level-index vector pair of some form. Excluding the array for the coefficients, the total memory footprint of the new data structure then merely amounts to

\[ 2 \cdot M \text{ bytes}, \quad M \text{ being the number of } \tilde{W}-\text{spaces,} \quad (4.14) \]

because \( k \) and \( n \) are small enough to be stored in one byte each. The full algorithm arising from these rules is shown in Alg. 5 at a reasonable level of abstraction. Its
4. Co-Design, Part II: Presenting Solutions

By interleaving coefficients of subspaces that are hierarchically dependent in dimension 0, a sequence of 1-D bfs trees is formed in which affected coefficients of the evaluation are clustered. This process is shown in Fig. 4.8a, where subspaces $W(\ast,3)$ together form space $\tilde{W}(3,3)$. The traversal order indicated for the $\tilde{W}$-spaces in Fig. 4.8b complies with the iterator presented for evaluation in Sect. 4.2.2. It ensures optimal calculation of the tensor product, and it further allows the systematic identification of the one affected bfs tree per $\tilde{W}$-space by means of a single integer $i$ (here in binary format, also cf. Fig. 4.8a). When proceeding to the next $\tilde{W}$-space, index $k$ denotes the dimension that is refined next (all dimensions $j < k$ are “reset”).

Compact form already indicates that its porting to other platforms – even GPUs – is rather straightforward. The algorithm considers grids without boundaries, but only the tensor product calculations and some offsets change in its extended form. It is further noteworthy that the index $i$ appearing in the algorithm as well as in both parts of Fig. 4.8 elegantly encodes all components of the index vector. The pseudorecursive approach takes advantage of this compressed format, too, as this tweak is what lowers the resource consumption enough to allow for a high performance GPU implementation of the algorithm in the first place. To conclude the description of the new data structure, a comparison of the pseudorecursive and the new hybrid layout is shown in Fig. 4.9. The two layouts are visually compared at the example of a small 3-D sparse grid with boundaries. The illustration furthermore gives an impression of the clustering of affected coefficients, which is particularly pronounced for the coefficients associated with boundary grid points.
Algorithm 5 The full evaluation algorithm for the new hybrid approach is shown for a grid without boundaries. The incremental tensor product computation is left out for simplicity, as are several optimizations related to the computation of sums. Input vectors \( \vec{n} \) and \( \vec{k} \) have length \( M - 1 \) and describe the transitions between the \( W \) of the sequence. Parameter \( n_0 \) relates to the first \( W \). Range slicing of vectors is done as usual, i.e., the second index is not part of the range.

```
1: function EVALHYBRID(\( \vec{x} \), \( \vec{\alpha} \), \( n_0 \), \( \vec{n} \), \( \vec{k} \))
2: \( s \leftarrow \) computeEval1dStencil(\( x_0 \), \( n_0 \))
3: \( \vec{l} \leftarrow \vec{r} \leftarrow \) getRootLevelVector()
4: \( i \leftarrow 0 \)
5: \( p \leftarrow \) computeTensorProduct(\( l[1 : d] \), \( i \), \( x[1 : d] \))
6: \( \text{result} \leftarrow p \ast \text{EVAL1DGENERAL}(s, n_0, 0, \vec{\alpha}) \)
7: \( \text{offset} \leftarrow 2^{n_0} - 1 \)
8: for all \((n, k) \in \vec{n} \times \vec{k}\) do
9: \( i \leftarrow i >> \sum_{j=1}^{k-1}(l[j] - r[j]) \)
10: \( i \leftarrow 2 \ast i + \) descendRight(\( l[k] \), \( x[k] \))
11: \( l[0 : k] \leftarrow r[0 : k] \)
12: \( l[k] \leftarrow l[k] + 1 \)
13: \( \text{size1d} \leftarrow 2^n - 1 \)
14: \( p \leftarrow \) computeTensorProduct(\( l[1 : d] \), \( i \), \( x[1 : d] \))
15: \( \text{result} \leftarrow \text{result} + p \ast \text{EVAL1DGENERAL}(s, n, \text{offset} + i \ast \text{size1d}, \vec{\alpha}) \)
16: \( \text{offset} \leftarrow \text{offset} + \text{size1d} \ast (2 - d + \sum_{j=1}^{d-1} l[j]) \)
17: end for
18: return result
19: end function
```
Figure 4.9.: The pseudorecursive (left) and the new hybrid layout (right) of a 3-D level 3 sparse grid with trapezoidal boundaries is shown. Each row depicts a 1-D bfs tree, and affected coefficients of the evaluation at point $\vec{x}$ are marked as empty red circles. Both layouts group coefficients with respect to the “slowest” dimension’s refinement level (cf. level vector components $l_z$ on the left). For the hybrid layout, the $\bar{W}$-spaces are indicated.

$\vec{x} = (0.4, 0.55, 0.9)$

evaluation point
4.3. The Core Tasks Solved

Performance Analysis

The presented data layout is a hybrid between a subspace-based storage layout (such as the one described in [94]) and the pseudorecursive data layout from [25]. In the latter work, it is shown that implementations specifically optimized for regular and truncated sparse grids outperform the fastest SASG implementations (e.g., see [69]), and it is further shown that the pseudorecursive approach achieves the best performance of all. The following discussion shall therefore primarily answer the question whether the performance observed for the new hybrid solution is closer to that of the subspace-based or that of the pseudorecursive approach. Comparisons with SASG implementations are irrelevant in that matter and are for this reason skipped.

Consider the cache usage of the three approaches. Figure 4.9 already gives an idea of the clustering of affected coefficients for the hybrid and the pseudorecursive layout. The coefficients seem slightly more scattered in the hybrid layout, an impression that is confirmed by the results in Tab. 4.1. The table gives the number of cache misses monitored for all three implementations during evaluation of level 8 sparse grids. As expected, the hybrid approach ranks second behind the pseudorecursive scheme. Nevertheless, especially for grids with boundaries (such as the one sketched in Fig. 4.9), the hybrid layout gets quite close to the pseudorecursive layout’s low cache miss rates. For grids without boundaries, hybrid and subspace-based approach are closer together, but the total number of cache misses is consistently lower for the hybrid approach. The contents of Tab. 4.1 are readings of the CPU’s performance monitoring units (PMU) acquired via PAPI. The test platform used in this experiment is an Intel Xeon E5-2680v2 dual ten-core Ivy Bridge server platform (32 kB L1 and 256 kB L2 data cache per core, 64-byte cache lines). In comparison to its predecessor, the SNB-EP platform used in all other experiments, it supports an extended set of PMU counters.

A three-way comparison of the three implementations helps to judge the performance of the new solution. To this end, Fig. 4.10 shows speedups of both the hybrid and the pseudorecursive variants over the subspace-based variant. The comparison is done for linear regular grids both with and without boundaries. Again, the choice of regular grids is only due to the limitations of the pseudorecursive approach, and no significant impact is to be expected when using the hybrid approach on general DASG. All three implementations are highly optimized and use all 16 cores in order to achieve maximum performance on the SNB-EP test platform. Additionally, they all rely on AVX intrinsics and 4-way vectorized handling of double precision floating point numbers. For the small test configurations on linear grids, the pseudorecursive variant is up to three times faster than the hybrid variant. For larger configurations the picture changes, and for some grid sizes the hybrid variant even wins the direct comparison.

The last comparison is one of different platforms. The performance of the hybrid approach is now examined for regular grids with boundaries on all three test platforms:

http://icl.cs.utk.edu/papi
4. Co-Design, Part II: Presenting Solutions

<table>
<thead>
<tr>
<th>$d$</th>
<th>DCL1</th>
<th>DCL2</th>
<th>DCL1</th>
<th>DCL2</th>
<th>DCL1</th>
<th>DCL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.022</td>
<td>0.001</td>
<td>0.044</td>
<td>0.001</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>3</td>
<td>0.546</td>
<td>0.002</td>
<td>0.501</td>
<td>0.002</td>
<td>0.370</td>
<td>0.001</td>
</tr>
<tr>
<td>4</td>
<td>2.569</td>
<td>0.101</td>
<td>2.170</td>
<td>0.115</td>
<td>1.759</td>
<td>0.041</td>
</tr>
<tr>
<td>5</td>
<td>6.393</td>
<td>3.349</td>
<td>5.481</td>
<td>3.335</td>
<td>4.277</td>
<td>1.273</td>
</tr>
<tr>
<td>8</td>
<td>55.622</td>
<td>36.196</td>
<td>48.240</td>
<td>38.821</td>
<td>33.307</td>
<td>11.045</td>
</tr>
</tbody>
</table>

Table 4.1.: L1 and L2 data cache misses (DCL1 and DCL2) are compared for the subspace-based, hybrid, and pseudorecursive evaluation schemes. The numbers in the table denote cache misses in millions, measured for 10,000 sequential evaluations of level 8 sparse grids with and without boundaries.
4.3. The Core Tasks Solved

The CPU implementation of the presented evaluation scheme for piecewise linear basis functions without boundaries is compared to the pseudorecursive variant from \[25\]. The subspace-based variant from \[93\] forms the baseline. All implementations use explicit 4-way vectorization (AVX) and 16 threads on the SNB-EP platform.

Figure 4.10.: The CPU implementation of the presented evaluation scheme for piecewise linear basis functions without boundaries is compared to the pseudorecursive variant from \[25\]. The subspace-based variant from \[93\] forms the baseline. All implementations use explicit 4-way vectorization (AVX) and 16 threads on the SNB-EP platform.

The SNB-EP CPU, the Nvidia Kepler GPU, and the Intel Xeon Phi. On the one hand, this study is proof of the hybrid scheme’s portability, and on the other hand, it shows that there is profit to the porting of the algorithms. The results of the comparison are shown in Fig. 4.11 again in form of speedups over the (fully optimized) subspace-based evaluation scheme running on the CPU. What meets the eye first is that

- all speedups are positive, ranging up to 17x for the MIC implementation,
- the CPU implementation scores for the small grids and those of low level,
- the Xeon Phi implementation is superior to the GPU implementation.

Besides the high degree of concurrency, the excellent performance observed for the accelerator platforms can be attributed to a hardware-supported *gather* operation. The AVX-512 instruction set available on the Xeon Phi is supplemented with *scatter* and *gather* operations for either 8 DP values or 16 SP values, making the Xeon Phi a real SIMD device. The regular AVX instruction set available on CPU architectures still lacks such operations, only its successor, the AVX2 instruction set, has finally been extended to support *gather*. Ever since the introduction of CUDA, Nvidia GPUs have been real SIMD devices. The SIMT programming paradigm facilitates scattered memory access, and generations of CUDA-capable architectures have refined the procedure in order to minimize the number of issued memory transactions.

Despite the different angles to the platforms’ SIMD features, the implementations for
Figure 4.11.: Implementations of the presented evaluation scheme on three platforms (CPU, GPU, and MIC) achieve excellent speedups for regular grids with boundaries when compared to the optimized CPU version of the subspace-based evaluation scheme from [93]. Both CPU versions run on the dual eight-core SNB-EP platform (4-way vectorization, 16 threads), the GPU version runs on an Nvidia Kepler K20x GPU, and the MIC version runs on the Intel Xeon Phi 7120p.

both accelerator platforms make use of explicit coalesced memory access. Generally, the Xeon Phi processes \( W_{AVX-512}^{(DP)} = 8 \) evaluations simultaneously in one thread, while the GPU synchronizes 32 evaluations across the threads of a warp. However, in order to exploit the affected coefficients’ clustering (see the lower levels of 1-D bfs trees in Fig. 4.9) in coalesced loads on the GPU, a scheme as presented in [25] is used. Therein, a warp’s threads temporarily team up in groups of four, carrying out four consecutive coalesced load operations related to four different evaluation tasks. In [25], this strategy leads to runtime reductions of up to 44% for certain grids. For the Xeon Phi evaluation kernel, the strategy is adapted in similar form: A thread taking care of evaluations 0–7 issues four \texttt{gather} operations in a loop, thus simultaneously loading four affected coefficients for evaluations 0 and 1, then 2 and 3, and so on.

4.3.3. UpDown

In this section, the focus is on the unidirectional principle once more. The performance study in Sect. 4.3.1 already addresses the general application of 1-D operators, and so this section rather revolves around algorithmic aspects of the UpDown scheme. First, the topic is the systematic generation of transposed operators as they are for example needed in the application of mass and stiffness matrices (cf. Sect. 3.2.3). Second, I
4.3. The Core Tasks Solved

discuss computationally optimal sequences for the scheduling of different $Up$ and $Down$ operators during the execution of the general $UpDown$ scheme. Third, I investigate the benefit of switching between the linear and the prewavelet basis in an attempt to reduce the computational load imposed by the unidirectional principle. To this end, I present timings obtained for the application of the bilinear forms defined by the $L_2$ scalar product and the Laplacian.

**Stencil Transposition**

In Sect. 3.2.3 two examples of scenarios are mentioned in which the transposition of 1-D operators becomes important. The obvious example is the application of symmetric mass and stiffness matrices, for which $Up$ and $Down$ are mutually transposed operators. More detailed discussions on this topic are found in [107, 117]. The other example is the application of an operator with respect to another basis, which requires the application of the transposed transformation when switching back to the first basis (cf. (3.25)). For further reading see [2].

The stencil formulations remind of shaders and can be considered small programs with an instruction set defined by the SubspaceStackBuffer interface. As such, they can be systematically processed, and since each instruction describes an invertible linear operation, the stencils can even be systematically transposed and inverted. Viewing them as matrix operations makes this a straightforward task. For example, the inverse of operation $xpay(j, k, a)$ ($j, k$ being the global indices of two basis functions $\phi_j$ and $\phi_k$) given by

$$\begin{pmatrix} 1 & \cdots & \cdots & \cdots & 1 \\ \cdots & 1 & \cdots & \cdots & \cdots \\ \cdots & \cdots & 1 & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 \end{pmatrix}$$

is simply $xpay(j, k, -a)$, and its transposed is $xpay(k, j, a)$. The inverse and transposed
of operation $axpy(j, k, a)$ given by

$$axpy(j, k, a) = \begin{pmatrix} 1 \\ \vdots \\ a \\ \vdots \\ 1 \end{pmatrix} \leftarrow j^{th}$$

are not directly contained in the set of instructions. The inverse is best expressed as the sequence $xpay(j, k, -1)$; $scale(j, 1/a)$, the transposed as the sequence $xpay(k, j, 1)$; $scale(j, a)$. The only thing left to do in the derivation of inverse and transposed operators is to reverse the instruction sequence. A small example for an inverse (linear dehierarchization) is found in Lst. 4.1 in Appx. A.1. A less trivial example, namely the transposed operator of the prewavelet transform, can be studied in Lst. 4.4.

### Operator Scheduling

The $UpDown$ scheme’s most intriguing properties are without doubt its elegance and its simplicity. Given the pair of $Up$ and $Down$ operators, implementing the general recursive call sequence sketched in Fig. 3.6a merely takes a few lines of code. But when aiming for the best possible performance, the adequate treatment of special cases such as the one in Fig. 3.6b is a must. Suddenly the distinction is not merely between $Up$ and $Down$ operators, instead, aspects such as the existence of boundaries (see discussion of the prewavelet basis in Sect. 3.2.3), or special treatment of one or more dimensions have a major influence on the optimal solution algorithm.

In the following, the efficiency of different $UpDown$ instances is compared by looking at the sizes of their $UpDown$ trees. To this end, function $op1d(A)$ is introduced for counting the number of (composite) 1-D operators sequentially computed when traversing operator $A$’s $UpDown$ tree.\(^{10}\) Consider the matrix $L_{\phi}(D)$ of the $d$-dimensional Laplacian operator \(^{(3.22)}\), and matrices $M(j)$ and $E(j)$ that apply the 1-D bilinear forms \(^{(3.19)}\) and

\(^{10}\) Note that the application of a 1-D operator only becomes expensive in conjunction with a sweep over the whole sparse grid. Hence, several 1-D operators that are consecutively applied to a loaded SubspaceStackBuffer are considered composite operators and are only counted as one by $op1d$. 

4. Co-Design, Part II: Presenting Solutions
4.3. The Core Tasks Solved

Listing 4.4: The C++ code snippet shows the transposed operator of the prewavelet transform in Lst. 4.3.

```cpp
void forward(int n, SubspaceStackBuffer* buffer) {
    if (n > 1) { // transposed: resolve redundancy on level 1
        buffer->xpay(2, 2, -1.0, 1, 1);
    }
    // top down traversal of the hierarchy
    for (int ℓ = 2; ℓ <= n; ++ℓ) {
        int i_max = 2^(ℓ-1);
        // 4.) transposed: temporary values for next level
        for (int i = i_max-1; i >= 2; i -= 2) {
            buffer->set(ℓ+1, 2i, ℓ, i);
            buffer->xpay(ℓ, i+1, -0.6, ℓ, i,);
            buffer->xpay(ℓ, i-1, -0.6, ℓ, i);
        }
        // 3.) LU decomposition: transposed backw. substitution
        for (int i = i_max-2; i >= 1; i -= 2) {
            buffer->scale(ℓ, i, e_i/2);
            buffer->xpay(ℓ, i+2, -0.4, ℓ, i);
        }
        buffer->scale(ℓ, i_max, bar_e_i_max/2);
        // 2.) LU decomposition: transposed forw. substitution
        for (int i = i_max; i >= 3; i -= 2) {
            buffer->xpay(ℓ, i-2, -d_i/2, ℓ, i);
        }
        // 1.) transposed: build RHS & resolve redundancy
        if (ℓ < n) { // no temporary values on highest level
            buffer->xpay(ℓ+1, 2i_max, -1.0, ℓ, i_max);
            buffer->xpay(ℓ+1, 2i_max-2, 0.5, ℓ, i_max);
            for (int i = 3; i < i_max; i += 2) {
                buffer->xpay(ℓ+1, 2i+2, 0.5, ℓ, i);
                buffer->xpay(ℓ+1, 2i, -1.0, ℓ, i);
                buffer->xpay(ℓ+1, 2i-2, 0.5, ℓ, i);
            }
            buffer->xpay(ℓ+1, 4, 0.5, ℓ, 1);
            buffer->xpay(ℓ+1, 2, -1.0, ℓ, 1);
        }
    }
}
```
in dimension \( j \in \mathcal{D} \), respectively. In 3-D, the general \textit{UpDown} scheme yields

\[
L_{\psi}(\mathcal{D}) = \left( \begin{bmatrix} E_{(x)}^\Delta + E_{(x)}^\nabla \end{bmatrix} \cdot M_{(y)}^\Delta + M_{(z)}^\nabla \cdot \left[ E_{(y)}^\Delta + E_{(y)}^\nabla \right] \right) M_{(z)}^\Delta \\
+ M_{(z)}^\nabla \left( \begin{bmatrix} E_{(x)}^\Delta + E_{(x)}^\nabla \end{bmatrix} \cdot M_{(y)}^\Delta + M_{(y)}^\nabla \cdot \left[ E_{(z)}^\Delta + E_{(z)}^\nabla \right] \right) \\
+ \left( \begin{bmatrix} M_{(x)}^\Delta + M_{(x)}^\nabla \end{bmatrix} \cdot E_{(y)}^\Delta + E_{(y)}^\nabla \cdot \left[ M_{(y)}^\Delta + M_{(y)}^\nabla \right] \right) M_{(z)}^\Delta \\
+ M_{(z)}^\nabla \left( \begin{bmatrix} M_{(x)}^\Delta + M_{(x)}^\nabla \end{bmatrix} \cdot E_{(y)}^\Delta + E_{(y)}^\nabla \cdot \left[ M_{(z)}^\Delta + M_{(z)}^\nabla \right] \right) \\
+ E_{(z)}^\nabla \left( \begin{bmatrix} M_{(x)}^\Delta + M_{(x)}^\nabla \end{bmatrix} \cdot M_{(y)}^\Delta + M_{(y)}^\nabla \cdot \left[ M_{(z)}^\Delta + M_{(z)}^\nabla \right] \right)
\]

(4.17)

while a customized algorithm for the application of the Laplacian leads to

\[
L_{\psi}(\mathcal{D}) = \left( \begin{bmatrix} M_{(y)}^\Delta + M_{(y)}^\nabla \end{bmatrix} \cdot M_{(z)}^\Delta + M_{(z)}^\nabla \cdot \left[ M_{(y)}^\Delta + M_{(y)}^\nabla \right] \right) E_{(x)}^\Delta \\
+ \left( \begin{bmatrix} M_{(x)}^\Delta + M_{(x)}^\nabla \end{bmatrix} \cdot M_{(y)}^\Delta + M_{(y)}^\nabla \cdot \left[ M_{(x)}^\Delta + M_{(x)}^\nabla \right] \right) E_{(y)}^\Delta \\
+ \left( \begin{bmatrix} M_{(x)}^\Delta + M_{(x)}^\nabla \end{bmatrix} \cdot M_{(y)}^\Delta + M_{(y)}^\nabla \cdot \left[ M_{(z)}^\Delta + M_{(z)}^\nabla \right] \right) E_{(z)}^\Delta.
\]

(4.18)

The general algorithm fails to distinguish between operators that consist only of either an \textit{Up} or a \textit{Down} part and those that require splitting. The customized algorithm on the other hand reorganizes the call sequence and honors the known diagonal structure of the operator \( E_{(j)} = E_{(j)}^\Delta \). One thus gets \( \text{op1d}(L_{\psi}(\mathcal{D})) = 3 \cdot 7 = 21 \) for the customized version (4.18), while the general version (4.17) leads to a total of \( \text{op1d}(L_{\psi}(\mathcal{D})) = 3 \cdot 14 = 42 \).

Class \textit{UpDownScheduler} is the result of careful analysis and can be configured to schedule a 1-D operator sequence of optimal length. The general \textit{UpDown} scheme depends on a single 1-D operator that is subject to a splitting into \textit{Up} and \textit{Down} operators, e.g., see mass matrix (3.21). Setting up the scheduler for this case requires one instance of class \textit{UpDownConfig} from Lst. 4.5\footnote{Listing 4.5 is simplified. The actual code can handle composite 1-D operators such as for instance \( E_{\psi}(k) = (H_{\psi}(k))^T \cdot E_{\psi}(k) \cdot H_{\psi}(k) \cdot k \in \mathcal{D} \) from (3.27d). \( E_{\psi}(k) \) propagates up and down the hierarchy and must therefore be of type \textit{UpDownJoint} to be scheduled after all \textit{Ups} and before the first \textit{Down}.} with the operator type set to \textit{UpDownConfig}. However, \textit{UpDownScheduler} also covers all special cases encountered in the context of this thesis, including the Laplacian (3.22) that depends on two different 1-D operators. For the \( L_2 \) scalar product in the Laplacian, the same \textit{UpDownConfig} as described for the mass matrix can be used. The energy scalar product is a mere scaling operation, i.e., the respective \textit{UpDownConfig} should have operator type \textit{Up} (or \textit{Down}). The scheduler interprets the given configuration and schedules 1-D operators in the following order:

1.) pure \textit{Up} operators: \textit{Up}

2.) the \textit{Up} part of operators subject to splitting: \textit{UpDownSplit}

3.) combined \textit{Up} and \textit{Down} operators: \textit{UpDownJoint}

4.) the \textit{Down} part of operators subject to splitting: \textit{UpDownSplit}
Listing 4.5: Class `UpDownConfig` specifies how the 1-D operator `op` is to be applied to the grid by the `UpDown` scheduler. Generally, `op` would be classified as `UpDownSplit`, if splitting is necessary and `forward` and `backward` are implemented. Special cases, in which only one of the two is implemented, are identified via `Up` and `Down`. If no splitting exists (cf. $E_\psi$ in (3.27d)), `UpDownJoint` needs to be used. Boundaries can cause the operator type to change, and so it makes sense to separately store a second operator type.

```cpp
struct UpDownConfig {
    enum class OperatorType : int {
        UpDownJoint = 0, // operators not split
        Up = 1, // pure Up operator
        Down = 2, // pure Down operator
        UpDownSplit = 3 // operators split into Up and Down
    };

    OperatorType opTypeInner; // used in no boundary case
    OperatorType opTypeBoundary; // used in boundary case
    msg::op::Operator1d* op; // the 1-D operator
};
```

5.) pure `Down` operators: `Down`

Finally, the class `UpDownScheduler` supports a parallel mode that lets the user take advantage of concurrent evaluation of the `UpDown` tree’s branches. In (4.18) for instance, it is obvious that each summand of the global sum can be computed independently. A similar but fully automated parallelization scheme of the `UpDown` algorithm based on OpenMP tasks is presented in [21, 107]. In contrast, the `UpDownScheduler` merely makes a suggestion how the `UpDown` tree could be decomposed for concurrent processing. It is still the programmer’s responsibility to dispatch the arising subtasks explicitly to different threads or processes via the scheduler’s parallel interface. The benefits of this approach are addressed in Sect. 5.3 where the optimized `UpDown` schemes are used for solving PDEs.

**Performance Results**

Instead of repeating the extensive tests for general 1-D operator application performed in Sect. 4.3.1 I directly focus on the algorithmic optimizations related to sparse FE operators. Recall that the approximation spaces spanned by the hat function and prewavelet bases are identical for DASG. With the fast basis transformations from Sect. 4.3.1 the sparsity of the prewavelets’ FE operators can be exploited for calculations in the linear
4. Co-Design, Part II: Presenting Solutions

Figure 4.12: The computation of the $L_2$ scalar product’s bilinear form via $UpDown$ is examined on regular grids without boundaries. The plot’s baseline is the matrix application in the hat function basis. The blue bars indicate speedups of a variant that switches forth and back between the prewavelet and the linear basis. The green bars show the comparison with a purely prewavelet-based implementation without basis switches.

Proof that the effort of switching to another basis is worthwhile is shown in Fig. 4.12 for the $L_2$ scalar product’s bilinear form (3.21), and in Fig. 4.13 for the Laplacian’s bilinear form (3.22).

In Fig. 4.12 the baseline is the application of $M_\phi(D)$ according to (3.26a). The blue bars indicate the speedup obtained when switching temporarily to the prewavelet basis and applying $M_\phi(D)$ according to (3.26c). The green bars refer to directly applying matrix $M_\psi(D)$ in the prewavelet space without performing a basis change. For the baseline’s $UpDown$ scheme, one gets $op1d(M_\phi(D)) = 2(2^d-1)$. Applying $M_\psi(D)$ directly in the prewavelet basis leads to $op1d(M_\psi(D)) = d$, switching to the prewavelet basis and back

Of course, another benefit is that the FE matrices are better conditioned for the prewavelet basis. This, however, can only be exploited when actually solving in the prewavelet space.
4.3. The Core Tasks Solved

Figure 4.13: Analogously to Fig. 4.12, the application of the Laplacian via *UpDown* is examined on regular grids without boundaries. The plot’s baseline is the matrix application in the hat function basis. The blue bars indicate speedups of a variant that switches forth and back between the prewavelet and the linear basis. The green bars show the comparison with the application of the Laplacian in the prewavelet space.

adds 2d applications of 1-D operators. This factor 3 can even be traced in the speedup numbers given in Fig. 4.12. By and large, the speedups ranging up into the hundreds (and even thousands for large problems) are solid evidence that taking advantage of the prewavelets’ degenerated *UpDown* tree is worth the effort.

In Fig. 4.13, the same analysis is exercised for the Laplacian operator. The comparison’s baseline is the direct implementation of (3.27a), leading to $op1d(L_\phi(D)) = d(2^d-1)$. Applying $L_\psi(D)$ directly in the prewavelet basis triggers $op1d(L_\psi(D)) = d^2$ 1-D operators (green bars). Switching temporarily to the prewavelet basis according to (3.27d) again adds 2d operators (blue bars). The speedups are less pronounced than for the mass matrix before. This is because the d *UpDown* trees making up $L_\phi(D)$ are only of half size compared to the tree of the mass matrix before. Furthermore, the dominating term
4. Co-Design, Part II: Presenting Solutions

in \( \text{op1d}(L_\psi(D)) \) is quadratic, which reduces the relative cost of the basis changes in the hybrid variant. In addition, the composite operator for the energy scalar product becomes slightly more costly in the prewavelet basis, as it actually consists of three operators (counted as one). Nevertheless, the speedups still range up into the hundreds and confirm once more that investing in basis changes is worthwhile.

4.3.4. Summary of Results

The DASG data structure described in Sect. [4.2] was put to test in several disciplines. The implementation passed all tests with flying colors – not only with respect to performance.

As for hierarchization, the formerly fastest published implementation [24] is beaten in its prime discipline on regular grids, owing to a more refined grid traversal and vectorization scheme (see results of separate tests in Figures 4.5 and 4.6). In combination, both optimizations lead to general speedups of at least 2x, with peaks at around 8x observed for some test cases. Another advantage of the new vectorization scheme is that it is freed from an inconvenient limitation of the old scheme, which yields incorrect results when the \( d \) dimensions are not processed in a certain order. Because of this enhancement, algorithms based on the general unidirectional principle can now take advantage of the same interfaces for 1-D operator applications.

For evaluation, a new coefficient layout specifically designed for DASG is shown to combine the strengths of other well-proven solutions. The improved clustering of affected coefficients in conjunction with the use of optimal algorithms leads to speedups over the fastest competitor of up to 13.5x on the CPU. In addition, the developed evaluation kernel has been gainfully ported to the GPU and the Xeon Phi. On average, the performance of both platforms is higher than on the CPU, with a noticeable increase for larger grids. On the Xeon Phi, a peak speedup of 17.2x over the “old” CPU implementation is observed. Although not demonstrated on the accelerator platforms, the classification setting briefly described in Sect. 5.1 underlines the relevance and quality of these results. In sparse grid classification, the \textit{multi-eval} operation needs to be completed as fast as possible in the computation of a regularized least-squares approximation.

For the third core task, the fast execution of the \textit{UpDown} scheme, the impact of algorithmic optimizations is examined. As demonstrated, clever basis changes have a huge return on investment, if they help to prevent the effects of the curse of dimensionality. For the application of common FE operators, the runtimes can in some settings be decreased by factors of several hundreds (even 1200 in one example), when basis changes are performed in order to take advantage of degenerated \textit{UpDown} trees. Although such high numbers speak for themselves, the potential of these optimizations only reveals itself in a real application. Hence, this missing piece is delivered in Sect. [5.3] where the algorithmic tweaks presented here lead to impressive performance results in the solution of the Black-Scholes equation.
4.4. Software Design

The final discipline – and this is also the bridge to what is coming next – is the software’s flexibility and usability. Several code snippets shown in this section are evidence of a programming interface for 1-D operators that facilitates the straightforward implementation even of complex functionality. An aspect not explicitly highlighted so far is the increase of programming productivity procured by the clear interface design. As detailed in the upcoming software section, many complex operators for several different types of basis functions have been defined in the course of this thesis, and as proven in the experiments, they all meet the requirements of scientific high performance software. Further proof that this is not where the potential of the developed software ends is given in the following section.

4.4. Software Design

The source code developed in the context of this thesis is exclusively written in the C++ programming language and makes use of features introduced by the C++11 Standard. Many ideas regarding code design and class organization are adopted from the SG++ toolkit\footnote{http://www5.in.tum.de/SGpp}, also because the plan to eventually integrate the code into the SG++ project code was already laid out at an early stage. The advantages of joining the SG++ project are manifold, as the SG++ code

- is already well-established in the community,
- is publicly available and maintained regularly,
- exhibits a high degree of flexibility and extensibility, and
- is commendable regarding documentation and usability.

It is to be expected that SG++ also greatly benefits from this step, since many of the new ideas for DASG data structures and algorithms presented in this chapter are generic enough to be profitable for SASG code as well. And this claim is by no means unfounded. After all, the code’s design principles (see Sect. \[4.1\]) are the result of careful analysis of all kinds of sparse grid implementations, and the principles’ validity therefore reaches beyond just DASG implementations. Hard proof of this is found in Sect. \[5.4\] where the idea of the key-value data structure for SASG from Sect. \[3.3.2\] is picked up and carried further. Excellent performance results are achieved which prove that the key-value approach really has the potential to break the predominance of the hash map in performance-critical settings. The fact that the interfaces of the DASG code have allowed for a successful and gainful integration of the SASG data structure within a very short time is solid evidence for the maturity and significance of the presented software solutions.
4. Co-Design, Part II: Presenting Solutions

The following paragraphs are intended to help programmers to understand the software’s underlying philosophy and to get started with the source code quickly. To this end, UML class diagrams give clues regarding the code structure and the relations between the core components. First, the similarities to the SG++ project are highlighted. Afterwards, the most relevant classes and interfaces are shown and extension points are explained.

4.4.1. Exchange with the SG++ Project

In [107], the SG++ project’s main design goals are formulated to be modularity, reusability, and the separation of data structures and algorithms. The project’s status in the sparse grid community proves that the developers succeeded in their ambitions and mastered the balancing act between flexibility and efficiency that is especially difficult in scientific codes. For this reason and because of the intention to merge the solutions in the long run, it has made sense to adopt the project’s philosophy up to the point of mirroring some of the high-level programming interfaces.

Quite obviously, the here-proclaimed shift of efficient scientific code towards more purposefully designed data structures causes a conceptual conflict with the design of universally applicable all-round solutions. Admittedly, the hash map has been and still is of great service to the sparse grid community, but the recent struggles of hash-map-based SASG codes regarding parallelization and performance improvement are clear symptoms that some changes are overdue. As for SG++, it is realistic to expect that the application of the design principles acquired in the course of this thesis will lead to a performance boost for some of the algorithms, given the project is ready for some rigorous changes in the code back-end. The front-end, i.e., the programming interface, will not necessarily be affected thanks to its careful and generic design.

The highest level of this generic design is illustrated in the UML class diagram in Fig. 4.14 and is almost directly adopted here. The central Grid class merely describes abstract entities consisting almost exclusively of meta information. The actual data structures for the different available grid types are encapsulated in implementations of the generic GridStorage interface (see also Fig. 4.15 of the following section). Grid manipulation and setup are done by the GridGenerator, including also the application of user-defined strategies for refinement and coarsening. Performance-critical components, such as implementations of the previously discussed core tasks, are formulated as operations. With factories automatically detecting a grid’s type and checking the availability of fitting operations, a highly intuitive and modular programming interface arises. Furthermore, the factory represents an obvious extension point for programmers interested in adding their own operations. The actual programming work associated with this step is described in

14 In the particular case of SG++, the reaction to this trend was the launching of side projects relying on the SoA data structure described in Sect. 3.3.3. This boosted the performance of sparse grid evaluation & Co. on modern architectures (e.g., see 69, 67), but the solutions are limited in their applicability and have not been merged back into the main project (yet).
4.4. Software Design

Following the example of the SG++ API, a sparse grid is represented by an abstract class `Grid` that does not per se offer any functionality to speak of. Depending on the grid type (identified by means of an enumeration type `GridType`), functionality can be dynamically added in form of exchangeable operations such as basis transformations (`OperationHierarchisation`), sparse grid interpolation (`OperationEvaluation`), or Finite Element operator applications (`OperationMatrix`). The actual grid implementation is hidden behind the `GridStorage` interface, and grid manipulation is done by an extra entity `GridGenerator`.

![Diagram of class structure](image)

the next section's examination of the lower-level interfaces.

4.4.2. Class Structure and Extension Points

The UML class diagram shown in Fig. 4.15 is complementary to the high-level view in Fig. 4.14. It targets developers interested in adding their own operations to the current set. Note that some entities in the diagram are simplified in order to not overload the diagram with information irrelevant for the task under consideration. This applies most of all to the various instantiations listed below the abstract interfaces `Operator1d`, `Sweep1dInterface`, `GridStorage`, and `SubspaceStackBuffer`. The sole point of these lists is to demonstrate that the respective implementations exist. Implementation details are not of interest here.

Taking advantage of the UpDown scheme for a new operation only directly involves classes `UpDownScheduler` and `UpDownConfig`. Consequently, relevant member variables and functions are given for these two classes. When defining a new operation, the developer's only responsibilities are
• providing 1-D operators in form of instantiations of the `Operator1d` interface,

• describing and linking these operators in instances of the `UpDownConfig` class,

• feeding the `UpDownConfig` instances into the `UpDownScheduler`.

If the `confSpecial` member of `UpDownScheduler` is not set, only the operator type configured in `confNormal` determines what kind of `UpDown` scheme is executed. Operator type `UpDownSplit` would lead to a scheme suitable for computing the $L_2$ scalar product’s bilinear form, whereas `Up` could be used for a simple hierarchization scheme. The application of the Laplacian requires the classification of a second operator (that for the energy scalar product), i.e., `confSpecial` needs to be set, too.

The bottom half of Fig. 4.15 describes the grid-specific components that make up the infrastructure used by `UpDownScheduler`. Their interaction is already addressed in Sect. 4.2.3 in particular in Lst. 4.2. Iterators make no appearance in the illustration as they are not subject to a unified interface valid for all grid types. Finally, observe the different shading of some entities. The reddish color marks components that exist in very similar form in SG++ but do not yet fully comply with the interfaces with which they are associated in the illustration. The blueish color identifies components originating from the work on regular sparse grids presented in [24]. The integration of this functionality into the new code base went smoothly, and only thanks to the code’s modular design it has been possible to isolate particular features such as coefficient layout, iterators, and vectorization schemes for comparison in the performance analyses in Sect. 4.3.1 and Sect. 4.3.2.

This description of the developed software can be concluded with a positive assessment. The design of SG++ has served as an excellent starting point for a novel set of software components and interfaces with great potential. If numbers alone are proof of the software’s usability and flexibility, then the large and versatile collection of functionality implemented for various platforms speaks for itself. On top, the software excels in terms of efficiency, besting competitive implementations in all disciplines. If any more proof of the software’s success is needed, it is found in 1), the effortless and consistent integration of the highly efficient “legacy” code for regular sparse grid from [23], and 2), the straightforward integration of a cutting-edge SASG implementation (see Sect. 5.4 for details) within very short time. With all these achievements, the software can be expected to serve as an instructive example for future projects, too.
4.4. Software Design

Figure 4.15.: The excerpt of the class hierarchy demonstrates the interaction between the central components. In order to optimally employ the unidirectional principle for a new operator, the programmer only needs to implement the Operator1d interface and configure class UpDownScheduler via class UpDownConfig. The slim parallel interface of UpDownScheduler further allows for concurrent processing of the UpDown tree’s branches.
5. Dimensionally Adaptive Sparse Grids in Action

In this chapter, I demonstrate the competitiveness of the new DASG software in relevant settings. Four applications stand witness to the developed solutions’ versatility and set new standards in more than one case for what is possible with sparse grids in non-distributed settings.

For warm-up, I briefly touch the problem of binary classification for data mining, however, exclusively for the computational challenge. Being inherently high-dimensional, such classification tasks are a natural match for sparse grids, and – interesting in this particular case – these tasks heavily rely on fast evaluation.

Second, I present a so far unpublished concept that links sparse grids to the real-time out-of-core rendering of large scientific data sets. The context is given by an interactive computational steering setting, in which fluent visual exploration of parameterized data sets is impossible without a hierarchical approach to the data. I show that the multi-level subspace splitting that underlies the sparse grid theory has beneficial properties for this setting, which are best exploited by a fast GPU implementation of the hierarchical transform. It will become clear from the discussion, that the DASG solutions meet all requirements, and thus, the stage is set for a first-time appearance of direct sparse grids in real-time visualization.

With the third application I return to the roots of sparse grids, namely the solution of partial differential equations. By applying the algorithmic tweaks presented for the UpDown scheme in Sect. 4.3.3 I solve the Black-Scholes equation for the numerical pricing of options in record time on a single CPU node. The first-time pricing of a seven-asset European basket call option therein represents a new record. In order to confirm the numerical quality of the computed results, I compare them with state-of-the-art SASG and CT simulations.

The last application follows a different direction. In an application of concepts, I reuse the design principles for efficient sparse grid software from Sect. 4.1 this time in order to derive a fast and flexible SASG implementation based on the extremely promising key-value approach mentioned in Sect. 3.3.3. The remarkable outcome of this experiment is proof of a broader applicability of the co-design approach.
5. Dimensionally Adaptive Sparse Grids in Action

5.1. Sparse Grids for Classification

Using sparse grids for classification and regression has a long tradition, and the importance of the approach has not decreased over time as is reflected by a selection of relevant publications between 2001 and today [48, 44, 107, 69]. The selected publications furthermore show that both the CT and SASG have been successfully used for classification tasks, which also renders DASG an option. In contrast to other problems solved with sparse grids, classification and regression rely on the fast completion of many independent subtasks, namely evaluations of the sparse grid function, which allows for highly parallel and even distributed processing (see [69]).

This gives me the perfect chance to apply and benchmark the solutions for evaluation from Sect. 4.3.2 in a real and relevant setting. Accordingly, this short application is to be considered an extended benchmark with no ambition of making a statement about the quality of sparse grids as a tool for machine learning. This topic is addressed in many different publications (cf. the ones mentioned before), where also the general competitiveness of the approach is proven in comparisons with other commonly used methods such as support vector machines, decision trees, or neural networks. Hence, I only superficially cover the necessary theory and then directly jump to a performance comparison with the currently fastest published (SASG-based) sparse grid classifier from [69, 66].

5.1.1. A Regularized Least Squares Approach for Classification

Classification and regression, both disciplines from machine learning, are closely related to each other. When done on sparse grids, even the same algorithm is used for both, although the tasks differ slightly. In regression, one seeks to recover a function from given samples, the \textit{training data}, in order to make an accurate (or at least reasonable) prediction for the function values of new data points. The setup in classification is similar, except the function values are class labels, and the goal is to classify new data points properly. Hence, classification on sparse grids is about finding separation manifolds between the set of available classes. In this benchmark here, I only consider a binary classifier which assigns values \{-1, +1\} to new individuals based on the recovered function’s sign. In the following, I formalize the classification process and look at the algorithmic challenge.

Let \( f : [0; 1]^d \rightarrow \mathbb{R} \) be a function that associates data points \( \vec{x} \) from the \( d \)-dimensional feature space with target values \( y \). The training data \( S \) is given by a set of \( M \) samples

\[
S := \{(\vec{x}_j, y_j) \mid f(\vec{x}_j) = y_j, 0 \leq j < M \} \subset [0; 1]^d \times \mathbb{R}.
\]

The task is to reconstruct an approximation \( f_N \in V_N \) of \( f \) from the data in \( S \), where \( V_N \) is some finite-dimensional function space. This is first done on sparse grids in [48].
by solving a regularized least squares problem

\[
    f_N := \arg\min_{g \in \mathcal{V}_N} \left\{ \frac{1}{M} \sum_{j=0}^{M-1} (y_j - g(\vec{x}_j))^2 + \lambda C(g) \right\}, \tag{5.2}
\]

where \( C \) is a smoothing operator. Minimizing the fidelity term in (5.2) ensures closeness to the training data, while minimizing the smoothness term generalizes the approximation to new, unseen data and can also help to prevent overfitting. The importance of either term in the overall minimization problem is controlled through the regularization parameter \( \lambda \), which can be determined for instance via cross-validation \[4\]. In the original approach from [48], gradient-based smoothing is performed with \( C(g) := \| \nabla g \|_{L^2}^2 \).

In [107], an operator based on the Euclidean norm of the surplus vector \( \vec{\alpha} \) is instead suggested specifically for hierarchical sparse grids. The approach is shown to lead to very similar results while reducing the computational cost considerably. It has meanwhile become common practice (e.g., see [69, 105]), which is why it makes sense to follow this approach here as well. For a sparse grid approximation \( f_N := \sum_{i=0}^{N-1} \alpha_i \phi_i \) with \( N \) points, (5.2) can be rewritten as a minimization problem

\[
    \arg\min_{\alpha_i, 0 \leq i < N} \left\{ \frac{1}{M} \sum_{j=0}^{M-1} \left( y_j - \sum_{i=0}^{N-1} \alpha_i \phi_i(\vec{x}_j) \right)^2 + \lambda \sum_{i=0}^{N-1} \alpha_i^2 \right\}, \tag{5.3}
\]

which can be turned into a system of linear equations (SLE)

\[
    \left( \frac{1}{M} B \cdot B^T + \lambda \cdot C \right) \vec{\alpha} = \frac{1}{M} B \cdot \vec{y}, \quad \text{where} \tag{5.4}
\]

\[
    B := (b_{i,j}) \in \mathbb{R}^{N \times M}, \quad \text{with} \ b_{i,j} := \phi_i(\vec{x}_j). \tag{5.5}
\]

5.1.2. Solution of the System of Linear Equations (SLE)

The SLE in (5.4) consists of \( N \) equations (one per grid point), and even though \( N \) is typically chosen to be several times smaller than \( M \) to avoid overfitting, it is not recommended to explicitly assemble the system matrix \( A \). Instead, it is recommended to use a fast iterative solver that only depends on the application of \( A \), which can be efficiently realized in a matrix-free scheme as described in the following paragraph. Here, the best choice for the SLE solver is the method of conjugate gradients (CG) [120], as it usually shows fast convergence. Note that this choice is only possible because \( A \) is symmetric positive definite.

A look at (5.5) reveals that the \( j \)-th element of vector \( \vec{\beta} := B^T \vec{\alpha} \) contains the value \( f_N(\vec{x}_j) \), i.e., applying \( B^T \) corresponds to evaluating the grid at the data points \( x_j, 0 \leq
5. Dimensionally Adaptive Sparse Grids in Action

Figure 5.1.: For a small example in 1-D, the problems related to parallelization of the operator $B$ are illustrated. $B^T$ corresponds to classical evaluation, i.e., the $\beta_i$ can be concurrently computed. To benefit from the optimality of the scheme, $B$ uses the same algorithm and just swaps the operands when computing the $\gamma_i$. However, if different threads distribute the contribution of $\beta_0$ and $\beta_1$, the write operations to array $\vec{\gamma}$ need to be synchronized.

With the algorithm from Sect. 4.3.2, this can be done fast and in parallel. $B$ corresponds to a “transposed evaluation” and can be done similarly efficient with almost the same algorithm. It is only necessary to swap the operands when building the sum in the evaluation formula (3.15) – at least in the sequential case. As illustrated at the example of a small 1-D sparse grid in Fig. 5.1, it gets slightly more difficult in the parallel case. Write conflicts can occur when access to the array $\vec{\gamma}$ is not synchronized.

The classifier used in the benchmarks below avoids the problem (and the synchronization overhead), as each thread operates on its own copy of $\vec{\gamma}$, and the copies are only summed up when the threads are joined.

5.1.3. Results

In the benchmarks, the artificial “checkerboard” data set is considered, which is defined by the function

$$f(\vec{x}) := \prod_{k=0}^{d-1} \begin{cases} -1, & \text{if } 1/3 < x_k \leq 2/3, \\ +1, & \text{else.} \end{cases}$$

The data set has already been used several times to examine the accuracy of sparse grid classifiers (e.g., see [47, 69, 105]). Although it is free of noise, it contains $3^d$ jumps that pose a challenge for sparse-grid-based learners.

For the performance comparison, the SASG-based classifier from [69, 66] is used in a shared memory configuration. It is currently the fastest published sparse-grid-based classifier, and it has achieved impressive solution times, especially in distributed settings with heterogeneous hardware. The classifier is highly optimized and nearly reaches peak performance on various parallel platforms, e.g., the test platforms considered in this the-
5.1. Sparse Grids for Classification

sis. However, a trade-off is made in the implementation, as the optimality of algorithms is sacrificed in order to be able to fully exploit the underlying hardware. The resulting streaming-based implementation uses explicit vectorization and does completely without conditional branching, even for the case of the modified linear basis functions (see [66]). The disadvantage of the choice of algorithms is, however, that the processing of large grids becomes very expensive.

In Tab. 5.1 the parameters of the classification benchmark are given. As can be seen, comparably small grid sizes between 18k and 36k grid points are chosen in order to prove the DASG classifier’s competitiveness in settings where the SASG classifier is known to be strong. In addition, the choice of smaller grids motivates the use of the DASG classifier for ensemble learning. The AdaBoost technique, for instance, greatly benefits from fast classifiers on small regular grids (e.g., see [68, 105]). The number of samples for training and testing is set to 200k, respectively. It is thus several times higher than the number of grid points, which avoids an overfitting of the data. This is confirmed by the numbers in the table, as training and testing accuracies do not deviate too much from each other. Note that in 5-D the accuracy decreases due to the more challenging problem (243 fields instead of 81) and the decision to not increase the number of grid points for the no-boundary case. In the case with boundaries, more grid points are spent, and a reasonable accuracy is maintained.

The outcome of the performance comparison is shown in Fig. 5.2. It seems surprising at first that the largest speedup over the SASG classifier is observed for the modified linear basis. This can be explained by the way the linear hat functions are implemented in the DASG evaluation algorithm. As discussed in Sect. 4.2, the DASG data structure supports flexible boundary configurations, even for single dimensions. Currently, one implementation serves to cover all cases (also homogeneous boundaries), thus costly conditional statements need to be evaluated. When explicit boundaries are present, even nested conditional branches are entered, which explains the performance drop observed for this case. Consequently, a separate treatment of the cases with and without boundaries would lead to an even faster implementation, in which the linear basis functions (without boundaries) would at least show the same performance as the modified linear basis functions. Note that conditionals are also the reason why the DASG implementation’s performance is relatively lower in the 5-D case. The SASG classifier gains ground in the comparison, as it still operates on grids of roughly the same size as in 4-D, and in contrast to the DASG classifier, it does not have to deal with an increasing number of conditional statements during grid traversal. Performance comparisons for evaluation shown in [25] demonstrate, however, that the used SASG evaluation falls behind as soon as the grid level is increased. The smaller speedups achieved in the boundary case are thus partially due to the boundary grids’ lower sparse grid level. By and large, the speedups between 7.5x and 43.8x are an excellent result (especially on the small grids), and they are proof of the DASG classifier’s competitiveness.
5. Dimensionally Adaptive Sparse Grids in Action

<table>
<thead>
<tr>
<th>4-D checkerboard</th>
<th>linear</th>
<th>modlinear</th>
<th>lin. with boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>#grid points</td>
<td>23297</td>
<td>23297</td>
<td>20481</td>
</tr>
<tr>
<td>reg. parameter λ</td>
<td>1e−6</td>
<td>1e−6</td>
<td>1e−6</td>
</tr>
<tr>
<td>max. CG iterations</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>accuracy train.</td>
<td>97.2%</td>
<td>93.3%</td>
<td>97.2%</td>
</tr>
<tr>
<td>accuracy test</td>
<td>94.9%</td>
<td>90.9%</td>
<td>95.8%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5-D checkerboard</th>
<th>linear</th>
<th>modlinear</th>
<th>lin. with boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>#grid points</td>
<td>18943</td>
<td>18943</td>
<td>36033</td>
</tr>
<tr>
<td>reg. parameter λ</td>
<td>1e−6</td>
<td>1e−6</td>
<td>1e−6</td>
</tr>
<tr>
<td>max. CG iterations</td>
<td>200</td>
<td>200</td>
<td>400</td>
</tr>
<tr>
<td>accuracy training</td>
<td>82.4%</td>
<td>79.1%</td>
<td>90.9%</td>
</tr>
<tr>
<td>accuracy test</td>
<td>75.7%</td>
<td>73.9%</td>
<td>84.9%</td>
</tr>
</tbody>
</table>

Table 5.1.: The setup of the checkerboard classification benchmark is shown. In order to avoid overfitting, the number of grid points in the used regular sparse grids is several times lower than that trainings and test data sets (200k samples for both in all experiments).

Figure 5.2.: The speedups of the DASG classifier over the SASG classifier are shown for the checkerboard benchmark in 4-D and 5-D. The sizes of the considered grids are given in Tab. 5.1.
5.2. Sparse Grids for Out-of-Core Visualization

In this second of four applications, I examine potential benefits of the sparse grid technique’s underlying multi-level subspace decomposition for an interactive computational steering setting. Previous works have come to the conclusion that the slowness of sparse grid interpolation excludes DASG and SASG implementations as viable options for the visualization of large data sets [127, 128]. Only the CT has so far managed to stand the real-time test, as it operates on full grids and can even take advantage of hardware-accelerated multi-linear (texture) interpolation. In spite of these findings, this application revolves around a concept that brings the direct sparse grid approach back into discussion for incremental out-of-core visualization by taking advantage of the fast hierarchical transforms presented in Sect. 4.3.1. Unfortunately, the integration of the presented concept into the software for computational steering developed in the context of [26] could not be accomplished before this thesis was finished. Yet, all components needed for the concept’s realization exist, and they are shown to satisfy the setting’s real-time constraint.

I enter the topic with a brief description of wavelets and the role they play in the out-of-core visualization of large data sets. I then explain a computational steering setting, whose specific requirements make out-of-core solutions such as those used in scientific visualization necessary. In the main part, I demonstrate that an approach trusting in a hierarchical data splitting based on sparse grids rather than wavelets has clear advantages in the realization of these solutions. For the related discussion of hierarchical transforms, I choose a wavelet perspective in order to stay close to the visualization context. The discussion includes details about an efficient GPU implementation of the sparse grid transform. In the results section, I propose two variants of sparse-grid-based hierarchical decompositions. I address the matter of approximation quality in experiments and make references to results of other works about sparse grid visualization. Finally, I show the outcome of the performance comparison between the GPU and the CPU implementation, which affirms that the GPU is a suitable target platform for the task at hand.

5.2.1. Out-of-Core Rendering and the Role of Wavelets

Out-of-core algorithms deal with problems whose size exceeds the capacity of the main memory and therefore depend on slower bulk storage or distributed data repositories. A field heavily in need of such algorithms is scientific visualization, where the data size knows almost no limits, and the user still desires visual feedback in form of images in real-time. The common way to bridge this gap between user expectation and machine capabilities is to resort to generating sequences of low-resolution images whenever the hardware fails to deliver the full data within a predefined response time. Then, the level of detail (LoD) in the image is gradually increased as soon as more pieces of data arrive.
For such a concept to work, a hierarchical representation of the data is needed. Simple subsampling of the data does usually not serve here, as this approach often leads to aliasing effects in the rasterization stage of image generation. Instead, filter kernels are typically applied to the data in order to smoothen it and thus eliminate the risk of undesired aliasing artifacts. Among these filters, wavelets hold the most prominent position (for general reading about wavelets refer to [33, 123, 9, 85, 86, 1], for an overview of wavelets frequently used in visualization see [31], e.g.). But wavelet filters are much more powerful and can be equally applied to generate LoD representations of large scientific data sets. Although wavelets are not used here in the end, the presented concept is inspired by the wavelet transform, and consequentially, a wavelet perspective serves best for its explanation. Up next is a short look at the one-dimensional discrete wavelet transform (DWT), before the targeted computational steering setting is described.

Similar to the one-dimensional multi-level subspace splitting used in the construction of sparse grids in Sect. 2.2.2, the wavelet transform also starts with a decomposition of the global function space (e.g., $L^2(\mathbb{R})$) into a nested sequence of subspaces $V_l$ and hierarchical complements $W_l$, such that

$$V_l \subset V_{l+1} \quad \forall l \in \mathbb{Z},$$

$$V_l = V_{l-1} \oplus W_{l-1} = V_0 \oplus W_0 \oplus \ldots \oplus W_{l-1}. \quad (5.7)$$

Under certain circumstances, wavelet theory states the existence of a scaling function $\phi$ (also called father wavelet) and a (mother) wavelet $\psi$, from which orthonormal bases $\{ \phi_{l,k} \}$ for the $V_l$ and $\{ \psi_{l,k} \}$ for the $W_l$ can be derived following

$$\phi_{l,k}(x) := 2^{l/2}\phi(2^l x - k) \quad \text{with} \quad \langle \phi_{l,j}, \phi_{l,k} \rangle = \delta_{j,k}, \quad (5.9)$$

$$\psi_{l,k}(x) := 2^{l/2}\psi(2^l x - k) \quad \text{with} \quad \langle \psi_{l,j}, \psi_{l,k} \rangle = \delta_{j,k}, \quad (5.10)$$

where $l, j, k \in \mathbb{Z}$. The $W_l$ are also called orthogonal complements, as the theory further states that

$$\langle \phi_{l,j}, \psi_{l,k} \rangle = 0 \quad \forall l, j, k \in \mathbb{Z}. \quad (5.11)$$

More details (no proofs) about the construction process of the ladder of subspace $V_l, l \in \mathbb{Z}$ and the scaling function $\phi$ are given in Appx. B.1. Here, the only other important thing to know is that $\phi$ is unique solution to the so-called scaling equation or dilation equation

$$\phi(x) = \sum_{k \in \mathbb{Z}} c_k \cdot \phi(2x - k), \quad (5.12)$$

a recursive equation that forms the connection between the $V_l, l \in \mathbb{Z}$ in the hierarchy. The $c_k$ in (5.12) are called filter coefficients (also scaling coefficients) and define the properties of the wavelet filter. For wavelet types with compact supports, only very few filter coefficients are in fact non-zero. This leads to efficient linear runtime schemes for the application of these filters, or in other words, an input signal can be hierarchically decomposed via the DWT in linear time. This process is often referred to as analysis. It is illustrated in Fig. 5.3 where the input vector is schematically split into approximation...
5.2. Sparse Grids for Out-of-Core Visualization

Figure 5.3.: On level $l \in \mathbb{N}^+$, a signal vector $\vec{s}^{(l)}$ of length $N = 2^l$ is analyzed, i.e., it is split into the signal’s approximation $\vec{s}^{(l-1)}$ and its corresponding detail $\vec{d}^{(l-1)}$ on level $(l-1)$, both being vectors of half length. The coefficients of vectors $\vec{s}^{(l')}$ and $\vec{d}^{(l')}$, $l' \leq l$ are weights associated with the basis functions $\phi_{l',k}$ and $\psi_{l',k}$ that are indicated next to the vectors.

and detail parts of half length. The inverse process is denoted synthesis and merges approximation and detail back into the original input signal.

One of the main strengths of the wavelet transform is that it retains the information about the location of high frequencies in the input signal. In contrast to an analysis completely conducted in the frequency space (as done by the Fourier transform), local features have only local influence in the transformed signal. Data compression is one of the fields that benefit most from this property: Much of the detail information is likely to fall below a predefined truncation threshold if the input signal is mostly smooth, and this even holds if the signal contains the occasional sharp edge. Obviously, sparse grids have some of these qualities, too. Same as wavelets, sparse grids capture high frequencies in the coefficients of the hierarchy’s higher levels. However, sparse grids interpolate data, whereas wavelets are filters that compute averages. This renders sparse grids unsuitable for the compression of integer data. But the picture is different for the task of out-of-core visualization described in the following section. There, the sparse-grid-based data splitting represents the better choice due to the computational steering setting’s specific requirements.

5.2.2. Computational Steering via Surrogate Models: Chances and Challenges

Computational steering of simulations remains one of the most challenging disciplines in computational science. Fast-growing simulations and increasing numbers of independent variables in optimization problems lead to large amounts of data to be analyzed. Interactive visual exploration is the only feasible option engineers are left with if tools for automated data analysis are unavailable. Nevertheless, even if an infrastructure exists that allows to take influence on the course of a running simulation (by varying a
boundary condition, perhaps), users are still faced with a clash of different timescales: A reasonable response time of an interactive system is within tenths of seconds, but it can take seconds, minutes, or even much longer until the results of the adjusted simulation are ready for display.

Surrogate models can help to overcome this problem to some extent. If the high-fidelity simulation is too computationally expensive, a model of reduced complexity is used in its stead. In [26], surrogate models are classified according to the following three categories:

- **hierarchical surrogates**: A lower-fidelity model (e.g., a coarser grid, simpler physics, etc.) is used to compute results faster.

- **reduced-order modeling**: After identifying the most important modes of the problem’s state space, a projected version of the PDE is solved.

- **data fit surrogates**: In a non-physics-based approach, available high-fidelity simulation results are used to approximate new results, typically via interpolation or regression.

### The Sparse Grid Surrogate Model (SGSM)

Subsequently, the focus is on data fit surrogates for problems depending on a moderate number of input parameters, and in particular on the sparse grid surrogate model (SGSM) first proposed in [29] and further elaborated in [27, 28, 26]. The simulation is therein considered a multi-dimensional scalar field

\[
u : \Gamma \times \mathcal{M} \to \mathbb{R},
\]

where \(\Gamma \subset \mathbb{R}^r\) is the simulation’s \(r\)-dimensional spatial domain, for which a cartesian full grid discretization with \(M\) grid points is assumed in the following. \(\mathcal{M}\) is the normalized \(d\)-dimensional space of input parameters given by

\[
\mathcal{M} := \{ \bar{\mu} \mid \bar{\mu} \in (0;1)^d \}.
\]

The high-fidelity result of the simulation at location \(\bar{\mu}\) of the parameter space will from now on be called a snapshot, and it is given by

\[
u(\bar{\mu}) := \nu(\bar{x},\bar{\mu}) \quad \text{with } \bar{\mu} \in \mathcal{M}.
\]

Analogously, let \(\tilde{u}(\bar{x},\bar{\mu})\) be the scalar field associated with an SGSM of simulation \(u\), and let the interpolated results \(\tilde{u}(\bar{x}) := \tilde{u}(\bar{x},\bar{\mu})\) be known as snapshots, too. In order to produce such interpolated results, a data fit surrogate model needs access to precomputed simulation data. The SGSM therefore uses an SASG to sample the parameter

---

The simulation can of course also be a vector field, such as would arise for instance from a flow simulation. Assuming a scalar field, however, simplifies the mathematical notation.
5.2. Sparse Grids for Out-of-Core Visualization

Figure 5.4.: The process of interpolation at parameter combination $\bar{\mu}$ is illustrated for a 2-D sparse grid surrogate model. Affected snapshots are depicted as small gray cubes within the sparse grid, and the interpolated snapshot $\hat{u}(\bar{\mu})(\bar{x})$ is depicted as red cube at the top.

Space $\mathcal{M}$ at $N$ locations $\bar{\mu}_j, 0 \leq j < N$, for which simulation snapshots $u(\bar{\mu}_j)(\bar{x})$ are then precomputed. As snapshots are stored on cartesian grids of $M$ points, the overall storage cost of the SGSM’s precomputed simulation data amounts to the space needed for storing $M \cdot N$ coefficients.

Storage allocation, data acquisition, and data preprocessing are typically not considered time-critical. Usually, these tasks are processed in the surrogate model’s offline phase when no user interaction takes place. Once accomplished, the model switches to the online phase and enables user interaction. At that point, a time constraint for fluent interaction with the system is introduced. Leaving aside the computation of new simulation results for the extension of the surrogate, the process most affected by such a constraint is data retrieval. For the SGSM, this process is graphically explained in Fig. 5.4. It differs from ordinary sparse grid evaluation only in that affected basis functions are weighed with snapshots of the simulation (depicted as gray cubes) instead of scalar coefficients. The upcoming discussions therefore exclusively revolve around the question how to enhance the evaluation of the corresponding formula:

$$\hat{u}(\bar{\mu})(\bar{x}) = \sum_{j=0}^{N-1} u(\bar{\mu}_j)(\bar{x}) \cdot \phi_j(\bar{\mu}).$$ (5.16)

Interpolating Snapshots in Real-Time

In [27], a multi-GPU configuration relying on five high-end devices of Nvidia’s Fermi generation (Tesla C2070) is found suitable for a parameterized CFD setting yielding 200 snapshots à 48 MB ($\approx 9.6$ GB total). The GPUs are hosted on five QDR-InfiniBand-connected nodes and hold equal-sized slices of the SGSM’s data in their main memory.
According to measurements, the overall time needed for the distributed evaluation of \((5.16)\) is around 20 milliseconds. This time is spent almost to equal parts in local computations and in the assembling of the interpolated snapshot via \texttt{MPI\_allgather}. While this result is well aligned with the limits specified for the response time of interactive systems (\(\approx 0.2\) seconds), increasing the data size clearly exposes the network as bottleneck: The assembly time goes up to 0.8 seconds for 150 snapshots à 192 MB, and furthermore, the data size of 28.8 GB already fills the GPUs’ main memory, disqualifying the approach for larger problems.

In [12, 26], the air flow around and through a large building is examined in another parameterized CFD simulation. Snapshot sizes therein range up to 1 GB, ruling out a GPU-based approach due to memory constraints. The sum \((5.16)\) is evaluated on several InfiniBand-connected CPU nodes, but only one master node is used for visualization. This permits replacing \texttt{MPI\_allgather} by a cheaper \texttt{MPI\_reduce}, but the time to completion still ranges up to 2 seconds despite this change, again clearly violating the real-time constraint. Further analysis reveals, that again most of the delay is due to network communication. But since bandwidth-limited CPUs are used for the interpolation (instead of high-bandwidth GPUs as in [27]), even the time spent in the local evaluation of \((5.16)\) already exceeds the response time limit.

In both settings, the self-imposed objectives regarding system response time are not fully met despite the use of cutting-edge hardware. Evidently, both scenarios call for scalable out-of-core solutions capable of handling large data sets without trying the user’s patience too much in interactive sessions. This again leads to hierarchical methods and an approach that successively gives access to the data’s levels of detail (cf. Sect. 5.2.1).

Figure 5.5 sketches how such a solution could look. The surrogate model uses a hierarchical transform (such as the DWT or sparse grid hierarchization) to decompose the snapshots in the offline phase. In the online phase, it immediately responds to requests by sending a small hierarchical prefix as coarse-level approximating of the data. Afterwards, it can start to successively interpolate hierarchical increments and deliver them asynchronously. After reception on the visualization side, a data format compatible with the visualization algorithms must be restored. To this end, the data is interpolated on a full grid using the fast hierarchical transforms. The details of this time-critical process are discussed in the following section. Note that correctness of the approach requires that the operators for the hierarchical transform and the interpolation of hierarchical increments according to \((5.16)\) are commutable. This is the case when the hierarchical transform is linear (which corresponds to constant filter coefficients).

The fast realization of the time-critical interpolation step completes a concept that fixes the shortcomings previously identified in the other approaches:

- The load on the network is reduced by the successive sending of hierarchically decomposed snapshots as smaller messages.
5.2. Sparse Grids for Out-of-Core Visualization

Figure 5.5.: The interplay of the surrogate model (left side) and the visualization (right side) is sketched. The surrogate hierarchically decomposes the snapshot data in a preprocessing step (e.g., via linear hierarchization). When a request is issued by the visualization, it is first answered with a hierarchical prefix of the data (approximation), before the remainder (detail) is delivered asynchronously level by level. On the receiving end, the inverse of the hierarchical transform is used to (re)construct full grid data for visualization.

- Interpolation and transmission of snapshots are not sequential anymore, as the hierarchical splitting facilitates asynchronous completion of both tasks.
- With a suitable hierarchical decomposition, the data increments can be chosen small enough to ensure their arrival and processing within the response time.
- If the resources on the render nodes permit it, hierarchical prefixes of all snapshots can be buffered locally for instantaneous response by the system.
- The concept scales to larger problem sizes, as the hierarchical approach to the data even allows for solutions that load from slower bulk storage.

5.2.3. The Hierarchical Transform on Full Grids

A comparison with the splitting induced by the DWT reveals why the multi-level subspace splitting is particularly useful for the computational steering setting under consideration. But for reconstructing full grid data suitable for visualization, it is necessary to step away from the intuitive approach that uses fast graphics hardware to accelerate
sparse grid evaluation. The process is too time-consuming, even when employing specifically optimized evaluation schemes such as those described in [93]. Instead, common wavelet algorithms lead the way and inspire a concept based on sparse grid hierarchization. Therefore, a wavelet perspective serves best in the explanations.

I first describe the general concept and its motivation. Afterwards, I discuss specifics of an efficient GPU implementation.

A Superior Hierarchical Splitting

Filtering and downsampling data are two of the strengths of the wavelet transform. However, the multi-level subspace decomposition yields a property that is even more important in the computational steering setting: It allows for a much more balanced splitting of the data into hierarchical increments.

Consider the schematic illustration of the 2-D wavelet transform in Fig. 5.6a. The signal $s^{(2)}(3 \times 3 \times 2 \times 2 = 64)$ coefficients) is recursively represented as collection of lower-level approximation and detail vectors, with areas symbolizing vector sizes. The multi-dimensional DWT is separable and uses a tensor product approach as well, i.e., one-dimensional filters must be applied in all of the $d$ dimensions to reach the next lower level of the hierarchy. As the size of the signal approximation is halved with every 1-D filter application, the approximation’s size diminishes with an exponential factor of $2^d$ when stepping from one level to the next in $d$-dimensions. The result is a very unbalanced splitting unsuitable for the scenario at hand, where the aim is to split a snapshot’s data evenly across several network messages.

The multi-level subspace splitting on the other hand allows for much finer tuning of the hierarchical increments’ sizes. The truncated sparse grid in Fig. 5.6b (truncation vector $\vec{c} = (3, 3)$, sparse grid level $n = 5$) has about the same size as the grid decomposed in Fig. 5.6a ($9 \times 9$ vs. $8 \times 8$), yet, the level-wise splitting (green dashed lines) yields a better balance between the increments’ sizes. Plus, it leads to one more increment than the DWT, even for this small example. The second splitting pictured in Fig. 5.6b is motivated by so-called $\sqrt{h}$ grids or two-scale grids. These grids are first introduced in [84] as low-cost variant for the CT, as they only require handling of $d + 1$ combi-grids. The name is due to the resolution-dependent choice of combi-grids which leads to a characteristic axis-aligned cut in the grids’ subspace tableau (cf. blue dotted lines in Fig. 5.6b). Examples demonstrating that the fine control over the increments’ sizes continues (and even improves) for larger grids are given in Tab. 5.2 of Sect. 5.2.4.

Both mentioned multi-level subspace splittings appear in publications about sparse grid visualization. In early works such as [127] [126] [125], large data sets are compressed via regular sparse grids and visualized using particle tracing and ray casting techniques. The data itself is considered smooth, and it partially originates from simulations on sparse grids. In [128], ray casting is used on truncated and $\sqrt{h}$ sparse grids for general (also...
5.2. Sparse Grids for Out-of-Core Visualization

Figure 5.6.: The hierarchical decompositions in 2-D are sketched for an $8 \times 8$ grid via the DWT (left) and a $9 \times 9$ grid via the multi-level subspace splitting (right). For the latter, two variants are shown: a level-wise splitting (green dashed lines) and $\sqrt{h}$ splitting encouraged by $\sqrt{h}$ grids often used with the CT. The decomposition defined by the multi-level subspace splitting is considerably better balanced already for the small example given.

Hierarchical Transforms for Interpolation

Now to the question how snapshot approximations can be reconstructed from incomplete hierarchical data, if conventional interpolation schemes are too slow. Shifting the perspective again helps. Wavelets are common for the compression of full grid data, as the analysis often outputs zero-details that can be stripped from the representation.\(^2\) Upon restoration of the original input via the inverse DWT, the dropped zeros need to be reintroduced. The same procedure can be used if the detail information – in form of coefficients for high-level subspaces $W_l$ – is simply not available yet. It suffices to

1.) reserve enough memory for the full grid,

\(^2\)In lossy compression schemes, very small values are considered to be zero as well.
5. Dimensionally Adaptive Sparse Grids in Action

2.) use the received hierarchical increments to initialize the low-level subspaces,
3.) set the coefficients of unavailable high-level subspaces to zero,
4.) run dehierarchization on all grid levels.

The resulting scheme represents the fastest way to obtain a coarse-level approximation of the data on the full grid. In contrast to conventional evaluation, no coordinate handling is required, since the interpolation implicitly takes place in the grid points. At the same time, one takes advantage of the sparse grid approximation theory, which postulates that the step-by-step reconstruction of the full grid data is cost-benefit-optimal with respect to the $L_2$ error.

Finally, GPUs suffer particularly from data formats with global data dependencies. In wavelet codes, this problem is usually addressed by grouping coefficients in tiles in order to localize data dependencies. A large full grid is then transformed into a full grid of tiles, which are essentially smaller full grids that can then be hierarchically decomposed. The difficulty is in “gluing” the tiles together consistently. Often, this can only be achieved if neighboring tiles overlap in more than one coefficient. While this can entail an immense increase of the data volume for some wider wavelet filters, consistency for the piecewise linear and polynomial functions is “already” ensured if only the boundary coefficients are replicated.

**Implementation on the GPU**

GPU programming is much more restrictive compared to the programming of CPUs. This concerns the versatility of the programming model, the availability of resources, and the high degree of parallelism needed for good performance. The good news is that the data structures and algorithms presented in Chapt. 4 avoid high-level software constructs that exclude their use on accelerator platforms. By and large, many ideas incorporated in the presented CPU implementation – especially regarding parallelization and vectorization – are compatible with graphics cards, too. But their re-implementation in OpenCL still takes effort, due to a necessary switch to explicit resource management and the SIMT programming paradigm (see introduction to OpenCL in Sect. 2.1.4).

Since many independent tiles of uniform shape and limited size need to be processed, all relative offsets of subspaces within tiles can be precomputed and cached in the GPU’s fast read-only constant memory. The subtasks arising for 1-D hierarchization are distributed to the GPU’s resources for concurrent processing as shown in Fig. 5.7 (boundaries are avoided in the illustration for clarity):

3 The only exception may be the hash map. But since even the CPU compute kernels suffer from costly look-ups, it is substituted with a proxy data structure, namely arrays buffering precomputed offsets for fast streaming. In the GPU code, the same principle is used.
5.2. Sparse Grids for Out-of-Core Visualization

Figure 5.7.: The GPU parallelization scheme for hierarchization in $y$-direction is sketched. A tile’s 1-D subspace hierarchies are assigned to 5 different work groups. Hierarchies of equal size qualify for processing by the same kernel (see labels). Parallelization is present in all layers: Tiles and subspace hierarchies are assigned to different work groups, work items of a work group split up the work within the hierarchies.

- Each work group takes care of one 1-D hierarchy of subspaces. If a hierarchy is too large to fit into a work group’s dedicated scratch pad memory, it is sliced up and assigned to several work groups (e.g., see work groups 3 and 4).

- A parameterized GPU kernel implementation can cover all sizes of 1-D hierarchies. Nevertheless, separate kernel launches are required if the kernel parameters differ. To this end, a preparatory step is carried out in which all 1-D hierarchies are grouped with respect to their implied kernel parameters.

- A work group’s work items process the coefficients in the hierarchy concurrently. On average, a work item takes care of more than one coefficient, in order to keep the number of idling GPU threads low when parallel work runs out on the lower levels of the hierarchy. A ratio of $1 : 4$ seems reasonable and is in line with contemporary GPU implementations of wavelet functionality.

As mentioned in the brief introduction to the language, OpenCL does not yet provide a way to synchronize work groups directly on the GPU. The synchronization point after each 1-D operator application is therefore implemented on the CPU side. Finally, the vectorization scheme used in the GPU kernel corresponds to the dimensional shift described in [24]. The advantage is that the scheme enables coalesced reads and writes from and to global memory also in case of large subspaces that are subject to slicing (cf. work groups 3 and 4 in Fig. 5.7). On the downside, computations are not in place.

In [129], a work item processes 4–8 coefficients (value known from private communication).
5. Dimensionally Adaptive Sparse Grids in Action

5.2.4. Results

The interest in this section is two-fold: The first question is how the splitting handles functions that do not comply with the usual smoothness assumptions made for sparse grids. Secondly, the suitability for real-time settings needs to be proven in benchmarks. To this end, timings obtained for the CPU and the GPU implementation are compared and related to the target setting.

Approximation Quality

According to sparse grid theory, the hierarchical surplus of smooth functions shrinks exponentially when advancing along the sparse grid level (cf. Sect. 2.2.2). The diagonal cut of the subspace tableau thus leads to a cost-benefit-optimal approximation of functions with respect to the $L_2$ norm. A reliable forecast regarding the approximation’s quality in general computational steering settings is, however, difficult – after all, the real world is usually not “smooth”. The only way to get an impression is through tests.

Consider the two test functions

\[
\begin{align*}
f(\vec{x}) &:= \cos(x_0 + 2x_0 x_1 + \frac{x_1}{4}) \cdot \cos(\frac{1}{4} + x_2) \quad \text{and} \quad (5.17) \\
g(\vec{x}) &:= \prod_{j=0}^{d} \frac{1}{\sigma_j \sqrt{2\pi}} \exp \left( -\frac{1}{2} \cdot \left[ \frac{x_j - \mu_j}{\sigma_j} \right]^2 \right) \quad \text{with} \quad \mu_j = \frac{1}{2}, \sigma_j = \frac{1}{16} \quad \text{for} \quad j \in \mathcal{D}, \quad (5.18)
\end{align*}
\]

that are “unpleasant” from the point of view of sparse grids, but might be encountered in solutions of PDEs in one form or another (see plots in Fig. 5.8). The signal-like function $f$ does not travel axis-aligned, and the waves even spread out due to the bilinear argument in the first cosine function. Larger mixed derivatives are thus to be expected for these dimensions, while the third dimension is separable and easier to handle. Interpolation of the Gaussian distribution $g$ is also studied in [107] in the context of likelihood estimation. It represents an ill-posed problem for sparse grids due to the locality of the feature and its steep flanks (especially for small variance $\sigma$).

In Tab. 5.2, the 2-D and 3-D variants of both functions are approximated with piecewise linear basis functions on cartesian grids with 4k–5k points. The table’s top half gives increment sizes and approximation errors obtained for the classical level-wise splitting. In the bottom half of Tab. 5.2, corresponding values obtained for the $\sqrt{h}$ splitting are listed. The given $L_2$ error $err(\cdot)$ of a reconstructed approximation (using SP floats) is computed with respect to a reference approximation of the analytic function on the same cartesian grid. Three things are immediately visible in the table:

- Better results are achieved for the “smoother” function $f$. 

5.2. Sparse Grids for Out-of-Core Visualization

<table>
<thead>
<tr>
<th>$l$</th>
<th>#coeffs</th>
<th>%</th>
<th>$\sum$ %</th>
<th>$\text{err}(f)$</th>
<th>$\text{err}(g)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>0.2</td>
<td>0.2</td>
<td>0.37</td>
<td>7.86</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>0.3</td>
<td>0.5</td>
<td>0.25</td>
<td>3.05</td>
</tr>
<tr>
<td>3</td>
<td>28</td>
<td>0.7</td>
<td>1.1</td>
<td>8.9e-2</td>
<td>2.29</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>1.5</td>
<td>2.6</td>
<td>3.2e-2</td>
<td>0.76</td>
</tr>
<tr>
<td>5</td>
<td>144</td>
<td>3.4</td>
<td>6.0</td>
<td>1.1e-2</td>
<td>0.39</td>
</tr>
<tr>
<td>6</td>
<td>320</td>
<td>7.6</td>
<td>13.6</td>
<td>3.1e-3</td>
<td>0.20</td>
</tr>
<tr>
<td>7</td>
<td>320</td>
<td>7.6</td>
<td>21.2</td>
<td>7.4e-4</td>
<td>5.3e-2</td>
</tr>
<tr>
<td>8</td>
<td>512</td>
<td>12.1</td>
<td>33.3</td>
<td>1.6e-4</td>
<td>1.8e-2</td>
</tr>
<tr>
<td>9</td>
<td>768</td>
<td>18.2</td>
<td>51.5</td>
<td>2.9e-5</td>
<td>6.5e-3</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>24.2</td>
<td>75.8</td>
<td>4.5e-6</td>
<td>1.1e-3</td>
</tr>
<tr>
<td>11</td>
<td>1024</td>
<td>24.2</td>
<td>100</td>
<td>4.7e-8</td>
<td>2.2e-7</td>
</tr>
</tbody>
</table>

$\sqrt{h}$ splitting

<table>
<thead>
<tr>
<th>$l$</th>
<th>#coeffs</th>
<th>%</th>
<th>$\sum$ %</th>
<th>$\text{err}(f)$</th>
<th>$\text{err}(g)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>256</td>
<td>6.0</td>
<td>6.0</td>
<td>0.57</td>
<td>3.01</td>
</tr>
<tr>
<td>1</td>
<td>125</td>
<td>2.9</td>
<td>8.9</td>
<td>0.22</td>
<td>4.40</td>
</tr>
<tr>
<td>2</td>
<td>244</td>
<td>5.8</td>
<td>14.7</td>
<td>1.5e-2</td>
<td>0.72</td>
</tr>
<tr>
<td>3</td>
<td>464</td>
<td>11.0</td>
<td>25.7</td>
<td>1.2e-3</td>
<td>3.4e-2</td>
</tr>
<tr>
<td>4</td>
<td>832</td>
<td>19.8</td>
<td>45.5</td>
<td>7.6e-5</td>
<td>1.7e-2</td>
</tr>
<tr>
<td>5</td>
<td>1280</td>
<td>30.3</td>
<td>75.8</td>
<td>4.5e-6</td>
<td>1.1e-3</td>
</tr>
<tr>
<td>6</td>
<td>1024</td>
<td>24.2</td>
<td>100</td>
<td>4.7e-8</td>
<td>2.2e-7</td>
</tr>
</tbody>
</table>

Table 5.2.: For sample settings in 2-D and 3-D, the table lists the sizes of hierarchical increments for the classical level-wise and the $\sqrt{h}$ splitting (green dashed lines and blue dotted lines in Fig. 5.6b, respectively). Percentages are shown to underline how well-balanced the splittings are. Furthermore, the reconstructed approximations’ $L_2$ error obtained for the linear nodal point basis on the full grid is given for test functions $f$ (5.17) and $g$ (5.18).
The level-wise splitting performs better than the $\sqrt{h}$ splitting.

Errors never completely vanish due to rounding errors in SP FP arithmetic.

For $f$, the error exhibits a steady decline, which seamlessly continues in the three-dimensional case. Good quality approximations are already obtained for 20–25% of the data. This confirms the result from [128] that adding one or two levels to the sparse grid representation improves the approximation considerably. The low-level approximations of the ill-posed problem $g$ demonstrate how the subsampling of local features can have global negative effects. For the $\sqrt{h}$ splitting, the accuracy starts to improve only from level 2, however, at least 45–65% of the data are needed for a reasonable accuracy of the approximation. For the level-wise splitting, the results improve only marginally.

As underlined by the results, the splittings perform best on smooth data. To get a visual impression how well they work when applied to photos, have a look at Appx. B.2.

Performance

Figure 5.9 compares the performance of the GPU and the CPU implementation for the reconstruction of several data sets in 2-D, 3-D, and 4-D. All data sets are roughly 1 GB in size and use single precision floating point numbers for the coefficients, which is common in visualization. The shown timings are for the linear transform, and they prove that a GPU implementation is worthwhile for at least two reasons:

- Except from one unlikely 2-D configuration, the GPU variant beats the fully optimized CPU variant (8-way AVX vectorization, 16 threads) in every comparison.
5.2. Sparse Grids for Out-of-Core Visualization

Figure 5.9.: Runtimes for different tile sizes in 2–4 dimensions are compared for the CPU test platform (dual socket SNB-EP NUMA system) and the GPU test platform (Nvidia Kepler K20x). Since the setting is related to visualization, single precision floats are used. On the SNB-EP platform, one can thus take advantage of 8-way vectorization via AVX and 16 threads.

- With the GPU capable of computing the transform, another bottleneck is avoided as only small hierarchical increments are sent over the PCIe bus to the GPU.

The study also reveals that the performance gain of tiling fades when choosing the tiles too small. In fact, the choice of the tile size should take into account the data as well. After all, the multi-level subspace splitting (in conjunction with the interpolating hat function basis) subsamples the data and thus potentially introduces undesirable artifacts.

Regarding the concept’s fitness for use in an integrated computational steering solution as described in [12], the overall system response time needs to be checked. In [27], interpolating and transmitting snapshots of 50–100 MB is measured to take less than 0.3 seconds. Adding the times listed in Fig. 5.9, this amounts to less than 0.5 seconds of overall delay. This worst case estimation still qualifies as interactive, and it holds also for large snapshots of up to 1 GB when split into sufficiently small hierarchical increments.

Last but not least, the whole approach could be extended with ideas from [73], where the hierarchization of full grids has recently been addressed. Available hierarchical coefficients could be written to their actual positions in the (zero-initialized) tile, before a delayed hierarchical transform is applied to the whole tile in full grid rather than subspace-based representation. As a consequence, arithmetic operations on the data would be less fragmented and memory access would be more coalesced. Considering the good results observed for the prewavelet-based splitting in Appx. B.2, another enhancement could be the use of actual wavelets on sparse grids as described in [52].
5.3. Sparse Grids for Solving Partial Differential Equations

The numerical solution of *partial differential equations (PDEs)* is one of the (if not the) most prestigious and challenging domains in scientific computing. In this application, I demonstrate the fitness of the software for the solution of parabolic PDEs in 2–7 dimensions. In order to prove the flexibility of the solutions and the competitiveness of the approach with other established techniques, I enter the field of financial mathematics and solve the Black-Scholes equation for the numerical pricing of options. The DASG implementation opens a new chapter in the book of sparse grids and the Black-Scholes equation, joining in a story of success started by solvers based on the CT (see works by different groups [112, 113, 83, 118, 41, 56]) and continued by direct sparse grid solvers relying on SASG (again, a list of works by different groups is [89, 109, 110, 107, 117, 67]).

Again, the primary interest is in the computational task at hand rather than the posed problem. The underlying theory is therefore distilled to the very essence and points to the fundamental literature for more extensive information on the topic of financial mathematics and option pricing in particular. In this spirit, I only show a brief validation of the simulation results’ numerical quality, in which I compare the results with accuracies and convergence rates mentioned in other works. The full attention is then directed to a problem-adapted static parallelization scheme. The scheme facilitates full exploitation of the CPU test platform’s parallel resources, which leads to impressive solution times. In a related discussion, I make suggestions how the principle can be extended to exploit node level parallelism on small clusters as well. Furthermore, the presented performance results are evidence of how solving the Black-Scholes equation in the prewavelet space leads to improved convergence and thus higher performance. To my knowledge, the implemented PDE solver is the first to successfully take advantage of the prewavelets’ strengths in a an implementation based on the direct sparse grid approach. Previous occurrences of prewavelets in sparse grid literature either refer to preconditioning [57, 58, 95], or they refer to solutions that are known to have flaws related to the discretization via SASG (e.g., see [37, 38]).

5.3.1. Application: Option Pricing

*Call options* are financial contracts that allow their holders to exercise the right to buy assets or financial instruments from their contract partner for a previously agreed-upon price at some point in the future. An option’s value consequently depends on the (stock market) performance of the asset, and so one generally speaks of *derivatives* (the options) and *underlyings* (the assets). Options are popular for several reasons. For one,
5.3. Sparse Grids for Solving Partial Differential Equations

Figure 5.10.: The hockey stick functions of call and put options are shown for maturity time $T$, strike price $K$, and an underlying’s price $S(T)$. Call options are only exercised if the price $S(T)$ is lower than the strike price $K$, because the payoff is zero otherwise. As the hockey stick functions for call and put options are symmetric to the axis at $S(T) = K$, there is only profit in put options for underlyings valued higher than $K$.

options – like other derivatives – are said to be “geared”, meaning that small movements in the underlying’s value can have large (positive) impact on the option’s value. Options can also help to guard or hedge against negative price fluctuations, for instance in volatile markets. As their holders are not obliged to exercise their right to buy, options allow traders to speculate with a minimized risk of loss.

In this application, only the simplest kind of call options, European options, are considered, for which the holder can strike only at maturity time $T$, buying for strike price $K$. With put options, a counterpart exists that grants a holder the right to sell assets for price $K$ at time $T$. Both kinds of options are characterized by their payoff functions $V_{\{\text{call, put}\}}$ at maturity time $T$,

\begin{align}
V_{\text{call}}(S, T) &:= \max\left\{S(T) - K, 0\right\}, \\
V_{\text{put}}(S, T) &:= \max\left\{K - S(T), 0\right\},
\end{align}

where $S(T)$ is the underlying’s price at time $T$. The payoff function is also commonly called hockey stick function due to its characteristic shape (cf. Fig. 5.10). The task in option pricing is now to determine an option’s fair price, i.e., the price at which the option should be sold in the present in order to minimize the risk of loss for both contract partners in the future. Note at this point that the process of pricing put options is essentially the same as for pricing call options (remember, the payoff function is simply mirrored). For the sake of compact representation of the essentials, put options are left out of the discussion from now on, and the focus is fully on European call options.

The crux of determining the price of an option is that the price depends on the future

\footnote{Many other more complex kinds of options exist, e.g., striking may be permitted at arbitrary or several specific times in the future, or strike prices may depend on formulas averaging the underlyings’ prices over a certain time.}
5. Dimensionally Adaptive Sparse Grids in Action

performance of the underlying, which is usually stochastic. A common first step in option pricing is therefore to model this behavior with a stochastic process \( X(t), t \in [t_0, T] \), given as the solution of a stochastic differential equation (SDE) of the form

\[
\begin{align*}
    dX(t) &= a(t, X(t))dt + b(t, X(t))dW(t), \\
    X(t_0) &= X_0,
\end{align*}
\]

(5.21)

where \( W(t) \) is a known Wiener process, i.e., a kind of stochastic process whose future values only depend on the present value. A solution \( X(t) \) to SDE (5.21) is also called Itô process. Unfortunately, analytical solutions for the option price only exist for very simple products. The pricing of more complex products such as baskets of options (simultaneously traded combinations of underlyings) calls for numerical methods. Boyle [15] introduced Monte Carlo (MC) methods for the pricing of European options, generating a large series of stock price trajectories \( S \) for the underlying, in order to obtain an estimate of the terminal option value \( V(S, T) \). Under the assumption of risk-neutral valuation, the fair price \( V(S, 0) \) is then derived as the expectation value of the payoff \( V(S, T) \) discounted with a risk-free interest rate \( r \), i.e.,

\[
V(S, 0) := e^{-rT}E[V(S, T)].
\]

(5.22)

MC simulations are still in use today because of their high flexibility (no assumptions are made regarding the stochastic process \( S \)), and just like in this case here, they are often employed to compute reference results. For other numerical methods readers are referred to the overview in [117].

5.3.2. The Black-Scholes Equation

The their 1973 seminal publications [13, 90], Black, Scholes, and Merton demonstrate for the first time how the fair price of a European option without early exercise feature can be determined under the following (idealized) market assumptions:

- “Frictionless” markets: No transactions costs or taxes are raised. Borrowing and short-selling are permitted, and trading takes place continuously in time.
- There is no chance of arbitrage.
- Markets are competitive, and assets are divisible.
- The risk-free interest rate is constant and known. It defines the rate for both, borrowing and lending.
- No dividends are paid.
5.3. Sparse Grids for Solving Partial Differential Equations

- The stock price is governed by a geometric Brownian motion, i.e., (5.23) is obtained from (5.21) for constant drift \( \mu \) and volatility \( \sigma \). The stock then follows a log-normal distribution, and the solution \( S(t) \) to SDE (5.23) is given by (5.24):

\[
\begin{align*}
\frac{dS(t)}{S(t)} &= \mu S(t)dt + \sigma S(t)dW(t), \\
S(t) &= S_0 \cdot \exp \left[ \left( \mu - \frac{\sigma^2}{2} \right) t + dW(t) \right].
\end{align*}
\]

(5.23) (5.24)

Under these assumptions, the Black-Scholes model builds upon the idea of a risk-neutral portfolio of call options \( V \) and a short position in the underlying \( S \). The randomness (and thus the risk) are completely eliminated from the model, leading to the conclusion that a risk-free interest bearing account yields the same growth as the risk-neutral portfolio. For baskets of several underlyings \( \vec{S} = (S_0, \ldots, S_{d-1}) \), correlations \( \rho_{ij} \) between them, volatilities \( \sigma_i \), the risk-free rate \( r \), and drifts \( \mu_i \), this leads to the multi-variate parabolic PDE (5.25) (see [119] for a detailed derivation):

\[
\begin{align*}
\frac{\partial V}{\partial t} + \frac{1}{2} \sum_{i,j=0}^{d-1} \sigma_i \sigma_j \rho_{ij} S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=0}^{d-1} \mu_i S_i \frac{\partial V}{\partial S_i} - rV &= 0, \\
\frac{\partial u}{\partial t} - \frac{1}{2} \sum_{i=0}^{d-1} \lambda_i \frac{\partial^2 u}{\partial z_i^2} &= 0 + 0 = 0.
\end{align*}
\]

(5.25) (5.26)

The multi-variate Black-Scholes equation (5.25) has no closed-form solution, but directly tackling it with numerical methods is not recommended either. It is numerically challenging due to undesirable properties, such as the correlations \( \rho_{ij} \), variable coefficients, and derivatives of first order. In [112], for example, several coordinate transformations and a principal axis transformation (PAT) are therefore suggested, which remove the problematic terms and reduce the problem of solving (5.25) to the problem of solving the simpler heat equation (5.26). Not only has the effectiveness of this approach been verified in numerical experiments, the concise reformulation of the problem further reduces both, the effort spent in computation and implementation, as is explained in [117]. The single steps of the transformation of the Black-Scholes equation into the heat equation can be studied in Appx. C.1.

The assumptions made for the Black-Scholes equation limit the applicability of the model considerably. Yet, it persists as the fundamental model for the pricing of options, and several extensions have been developed to account for some of its shortcomings. One such extension is a model that considers dividend payments, for instance described in [130]. In [39, 130] it is also explained, how a solver for European put options can easily be turned into one for American put options, by simply applying the early exercise constraint explicitly in every time step of the solution process (note that there is no benefit in the early exercise of American call options). Given these possibilities of extension and the context of this application, all considerations related to the presented solutions are restricted to European call options.
5. Dimensionally Adaptive Sparse Grids in Action

5.3.3. Finite Elements on Sparse Grids for Parabolic PDEs

Except from the choice of a DASG for discretization of the multi-dimensional space (and all implications this entails concerning algorithms and data structures), the given setup strongly resembles the solution approach successfully applied for SASG many times before (e.g., see [21, 22, 117, 67, 66]). Therefore, I take advantage of the knowledge about the solution process that has accumulated in these works and adopt the rough structure of the therein proposed Finite Element solver. A brief overview of the techniques applied for discretization in space and time, as well as the resulting meta-algorithm follows.

Finite Elements for Dirichlet Boundary Problems

Since the domain resulting from the coordinate transformations stretches from \(-\infty\) to \(\infty\) in all dimensions, a truncated version needs to be used in the computations. An overview of guidelines for reasonable truncation is found in [117]. Let the resulting domain be

\[
\Gamma := [S_{0}^{\min}, S_{0}^{\max}] \times \cdots \times [S_{d-1}^{\min}, S_{d-1}^{\max}] \subset \mathbb{R}^d. \tag{5.27}
\]

On \(\partial \Gamma\), Dirichlet boundary conditions are specified, which are considerably simplified owing to the transformation to the heat equation: Elimination of the Black-Scholes equation’s reactive term leads to conditions

\[
u(\vec{z}, \tau) = u(\vec{z}, 0), \quad \forall (\vec{z}, t) \in \partial \Gamma \times [0, T], \tag{5.28}
\]

which miss the dependence on the discount rate that is exhibited by the untransformed boundary conditions. The transformed conditions thus remain constant over time and require only one-time initialization (note that \(u(\vec{z}, 0)\) corresponds to the payoff \(V(\vec{S}, T)\) at maturity time, cf. Appx. C.1). This is described in detail in the following subsection.

In each time step, (5.26) needs to be solved for fixed \(\tau\). This is done directly in the hierarchical function space by applying the Finite Element Method. A variational formulation (or weak form) of the continuous problem is derived based on admissible test functions \(w \in H^1_0(\Gamma)\) (i.e., \(w \in H^1(\Gamma), w|_{\partial \Gamma} = 0\)):

\[
\langle \dot{u}, w \rangle + \frac{1}{2} \sum_{k=0}^{d-1} \lambda_k \left\langle \frac{\partial u}{\partial z_k}, \frac{\partial w}{\partial z_k} \right\rangle = 0, \tag{5.29}
\]

where \(\langle \cdot, \cdot \rangle\) denotes the \(L_2\) scalar product over the domain \(\Gamma\). For the discretization step, the hierarchical basis \(\Phi\) with \(V = \text{span}\{\Phi\}\) is split into the boundary basis functions \(\Phi^B = \{\phi_j^B, 1 \leq j \leq N^B \mid \phi_j^B|_{\partial \Gamma} \neq 0\}\), and those with homogeneous boundaries \(\Phi^I = \Phi \setminus \Phi^B = \{\phi_j, 1 \leq j \leq N^I\}\). \(V^I = \text{span}\{\Phi^I\}\) must represent a good approximation to the test function space \(H^1_0(\Gamma)\), in order to ensure a high quality numerical solution. The
5.3. Sparse Grids for Solving Partial Differential Equations

The weak form \( (5.29) \) is then projected onto \( V^I \) via a Ritz-Galerkin projection, leading to a linear system of equations of size \( N = N^I + N^B \)

\[
M\tilde{u} + \frac{1}{2}L_{\chi}\tilde{u} = 0, 
\]

(5.30)

where \( M \) and \( L_{\chi} \) are bilinear forms for the \( L_2 \) scalar product and a weighted Laplacian, respectively, given by

\[
M := (m_{i,j}) \in \mathbb{R}^{N \times N}, \quad m_{i,j} := \langle \phi_i, \phi_j \rangle = \prod_{m=0}^{d-1} \langle \phi_{m,i}, \phi_{m,j} \rangle, 
\]

(5.31)

\[
L_{\chi} := (l_{i,j;\chi}) \in \mathbb{R}^{N \times N}, \quad l_{i,j;\chi} := \sum_{k=0}^{d-1} \lambda_k \cdot \left\langle \frac{\partial \phi_{i,k}}{\partial z_k}, \frac{\partial \phi_{j,k}}{\partial z_k} \right\rangle \cdot \prod_{m \neq k} \langle \phi_{m,i}, \phi_{m,j} \rangle, 
\]

(5.32)

with \( \phi_i(\vec{x}) = \prod_{k=0}^{d-1} \phi_{i,k}(x_k) \in \Phi \) and \( \phi_j(\vec{x}) \) analogously. I now discuss the discretization in time and the derivation of the final system of linear equations.

**Discretization in Time and Solution of the System of Linear Equations (SLE)**

The very reason why sparse grids are such a good match for the option pricing problem is the general smoothness of the approximated payoff function \( V(\vec{S}, t) \), a property that also allows for desirable time stepping methods of higher order, such as the implicit second order Crank-Nicolson scheme. However, the smoothness assumption does not always hold, as the initial condition \( (5.33) \) is given by the discounted payoff at maturity time, which is a scaled version of the hockey stick function. It exhibits the characteristic \( (d-1) \)-dimensional kink and thus violates the smoothness assumptions of both, the space and the time discretization. As can be seen in \( (5.34) \), the violation equally occurs in transformed coordinates:

\[
V(\vec{S}, T) = e^{-rT} \cdot \max \left\{ \overline{S}(T) - K, 0 \right\}, 
\]

\[
u(\vec{z}, 0) = \max \left\{ K - \frac{1}{d} \sum_{i=0}^{d-1} \exp \left( \sum_{j=0}^{d-1} q_{ij} z_j \right), 0 \right\}. 
\]

(5.33)

(5.34)

Besides ensuring an appropriate spatial discretization (e.g., through high grid resolution or adaptivity), it has therefore proven useful to comply to the rules stated for pure parabolic problems in \[111\], i.e., a finite number of fully implicit initial time steps should be taken in order to smooth out rough data and achieve higher order convergence. The approach is commonly known as *Rannacher Smoothing* and has previously been applied for the option pricing problem, e.g., in \[39, 21, 117, 67\], and it is also employed here.

Let \( \delta \tau \in (0, T) \) be the constant time step size, and let \( \tilde{u}^{(k)} \) be the solution at time \( t^{(k)} = k \cdot \delta \tau \). For the initial implicit Euler time steps, where \( u^{(k+1)} := u^{(k)} + \delta \tau \cdot f(t^{(k+1)}, u^{(k+1)}) \) holds, \( (5.30) \) thus becomes

\[
(M + \frac{\delta \tau}{2}L_{\chi})\tilde{u}^{(k+1)} = M\tilde{u}^{(k)}. 
\]

(5.35)
5. Dimensionally Adaptive Sparse Grids in Action

For Crank-Nicolson, where \( u^{(k+1)} := u^{(k)} + \delta \tau \frac{f(u^{(k)}, u^{(k+1)}) + f(u^{(k+1)}, u^{(k+1)})}{2} \) holds, one gets

\[
(M + \frac{\delta \tau}{4} L_{\chi}) u^{(k+1)} = (M - \frac{\delta \tau}{4} L_{\chi}) u^{(k)}.
\] (5.36)

At this point the vectors \( \tilde{u}^{(k)} \) still have full length \( N = N^I + N^B \), i.e., the entries for the boundary functions in \( \Phi^B \) directly correspond to the Dirichlet boundary conditions. Since these entries are known, (5.35) and (5.36) can be transformed into smaller systems of size \( N^I \) that encode the boundary conditions into the right-hand side as described for an analogous SLE \( A \tilde{u} = \tilde{b} \) of size \( N = N^I + N^B \) in

\[
A \tilde{u} = \tilde{b} \iff \begin{pmatrix} A^I & A^B \end{pmatrix} \begin{pmatrix} \tilde{u}^I \\ \tilde{u}^B \end{pmatrix} = \begin{pmatrix} \tilde{b}^I \\ \tilde{b}^B \end{pmatrix}
\]

(5.37)

\[
\iff A^I \tilde{u}^I = \tilde{b}^I - A^B \cdot \tilde{u}^B = \tilde{b}^{rhs}.
\]

(5.38)

The final solver algorithm is given by Alg. 6. Remember from (5.28) that the boundary conditions are constant over time, i.e., the boundary part \( \tilde{u}^B = \tilde{u}^{(k)} |^B \) of the solution vector \( \tilde{u}^{(k)} \) is the same in every time step. This even allows to generically transform lines 5 and 10 (construction of the right-hand side) into

\[
\tilde{b}^{rhs} = \begin{pmatrix} M + \beta \cdot L_{\chi} \end{pmatrix}^I \tilde{u}^{(k)} |^I - \begin{pmatrix} M + \beta' \cdot L_{\chi} \end{pmatrix}^B \tilde{u}^B
\]

\[
= \begin{pmatrix} M^I + \beta \cdot L_{\chi} \end{pmatrix} \tilde{u}^{(k)} |^I + \begin{pmatrix} M^B + \beta \cdot L_{\chi} \end{pmatrix} \tilde{u}^B - \begin{pmatrix} M^B + \beta' \cdot L_{\chi} \end{pmatrix} \tilde{u}^B
\]

\[
=: A_{\beta}^I, \text{ cf. } (5.40)
\]

(5.39)

where the effect of the boundary on the interior \( D^I \) is constant over time and therefore precomputable. The resulting SLE is symmetric due to the structure of matrices \( M \) and \( L_{\chi} \), hence, the method of \textit{conjugate gradients} (CG) \[120\] represents a good choice for its solution. Finally, note the composition of the system matrices \( A \), which are all of the form

\[
A_{\beta} := (M + \beta \cdot L_{\chi}), \quad \text{with } \beta \in \left\{ 0, -\frac{\delta \tau}{4}, +\frac{\delta \tau}{4}, +\frac{\delta \tau}{2} \right\}.
\]

(5.40)

This linearity of the matrices \( M \) and \( L_{\chi} \) in \( A_{\beta} \) leads to a concise modular implementation, and it can be exploited in the parallel application of the system matrix, as is detailed in the next section.

5.3.4. Concurrent Processing of the \textit{UpDown} Tree

The computational hotspot in Alg. 6 is the solution of the SLE in lines 6 and 11, as the iterative CG solver repeatedly applies the system matrix \( A_{\beta} \). According to (5.40),
Algorithm 6 The resulting Black-Scholes solver uses Rannacher smoothing in the first \( n_{RS} \) time steps, before it switches to the Crank-Nicolson scheme for higher order convergence. In all time steps, operators only need to be applied to the inner grid \( V^I \).

1: function solveEuropeanCallOption(\( \vec{w}^{(0)}, \delta \tau, n_{RS} \))
2: \( \tau \leftarrow 0 \) \hspace{1cm} \triangleright \text{just include time } \tau \text{ for clarity}
3: \( n \leftarrow T \frac{\tau}{\delta \tau} \)
4: for \( 0 \leq k < n_{RS} \) do
5: \( \vec{b}^{\text{rhs}} \leftarrow M \vec{u}^{(k),I} - (M + \frac{\delta \tau}{2} L_{\vec{\chi}})^B \vec{u}^B \) \hspace{1cm} \triangleright \text{implicit Euler time steps}
6: \( \vec{u}^{(k+1),I} \leftarrow \text{solveSLE} \left( (M + \frac{\delta \tau}{2} L_{\vec{\chi}})^I, \vec{b}^{\text{rhs}} \right) \) \hspace{1cm} \triangleright \text{solve SLE of reduced size}
7: \( \tau \leftarrow \tau + \delta \tau \)
8: end for
9: for \( n_{RS} \leq k < n \) do
10: \( \vec{b}^{\text{rhs}} \leftarrow (M - \frac{\delta \tau}{4} L_{\vec{\chi}})^I \vec{u}^{(k),I} - (M + \frac{\delta \tau}{4} L_{\vec{\chi}})^B \vec{u}^B \) \hspace{1cm} \triangleright \text{Crank-Nicolson time steps}
11: \( \vec{u}^{(k+1),I} \leftarrow \text{solveSLE} \left( (M + \frac{\delta \tau}{4} L_{\vec{\chi}})^I, \vec{b}^{\text{rhs}} \right) \) \hspace{1cm} \triangleright \text{solve SLE of reduced size}
12: \( \tau \leftarrow \tau + \delta \tau \)
13: end for
14: return \( \vec{u}^{(n)} \) \hspace{1cm} \triangleright \text{return solution at time } \tau = T
15: end function

This in turn triggers \textit{UpDown} runs for \( M \) (5.31) and \( L_{\vec{\chi}} \) (5.32) on the inner grid. One or two more such runs (depending on the time discretization) per time step come with the construction of the right-hand side following (5.39). The \textit{UpDown} scheme on the inner grid thus becomes the critical component of the solution process.

Let again count \( op1d(A) \) the 1-D operators sequentially computed when applying operator \( A \) via \textit{UpDown} (cf. Sect. 4.3.3). Matrix \( M \) (5.31) applied to a piecewise bilinear grid leads to an \textit{UpDown} tree as shown in Fig. 5.11a and yields \( op1d(M) = 6 \). For split level 1 (cf. Fig. 5.11b), processes \( p_0 \) and \( p_1 \) can perfectly split up the work and independently process three operators each. Split level 2 again reduces the work per process, but it introduces redundant computations causing the overall work to increase (cf. Fig. 5.11c).

As briefly mentioned in Sect. 4.3.3, the class \textit{UpDownScheduler} offers support for the concurrent processing of an \textit{UpDown} tree’s branches. It accepts a split level and a processor number as indicated in Fig. 5.11 which it uses to identify the subtree to be processed. Thus, the work associated with large trees (think about high-dimensional settings) can be evenly divided among different processors. What is more, the approach scales to several operators. Take the example of the system matrix \( A_\beta \) (5.40): Two schedulers can split the \textit{UpDown} trees of \( M \) and \( L_{\vec{\chi}} \) into equal-sized subtrees for parallel processing by a pool of threads. Table 5.3 lists \( op1d\)-counts for \( M \) and \( L_{\vec{\chi}} \) on linear and prewavelet grids with and without boundaries. The numbers give an idea how much the

\[ \text{Table 5.3 lists } op1d\text{–counts for } M \text{ and } L_{\vec{\chi}} \text{ on linear and prewavelet grids with and without boundaries. The numbers give an idea how much the} \]

---

7 Not to mention variants of the Black-Scholes equation relying on four or more FE operators. The parallel mode of \textit{UpDownScheduler} is especially useful in such complex settings, as it can help to define and dispatch equal-sized subtasks for parallel processing.
critical path length varies between minimum and maximum split levels. Note further that it is \( op1d(A_\beta) = \max \{ op1d(M), op1d(L_\lambda) \} \) in the concurrent case.

In [21, 107], the UpDown scheme is elegantly parallelized using the OpenMP task concept. When available, several tasks concurrently descend in the tree and process the subtrees in parallel. UpDownScheduler does almost the same thing when used for a single operator on a shared memory system. However, besides being able to balance the load for several operators, it can also help to introduce NLP in an application. And this makes sense, after all, the application of 1-D operators already takes advantage of TLP and ILP, and the strong scaling results in Sect. 4.3.1 suggest memory bandwidth boundedness for thread numbers \( T > 8 \). Of course, the cost of the finalizing reduce operation must always be taken into account in the cost-benefit analysis.

### 5.3.5. Results

First, I list the details of the simulations (simulation parameters, hardware, etc.). Then, I look at the quality of the results and the simulations' timings. Last, I give a short summary and an outlook.

**Simulation Setup**

All simulations discussed in the following solve the problem of numerically pricing European basket call options. The parameters are chosen as follows:
5.3. Sparse Grids for Solving Partial Differential Equations

\[
\begin{array}{cccccc}
\text{dim} & \{M^I_\phi, M_*\} & \{L^I_{\tilde{X}_\phi}, L_{\tilde{X}_*}\} & \{M^I_\psi\} & \{L^I_{\tilde{X}_\psi}\} \\
2 & \text{sequential} & 6 & 6 & 2 & 4 \\
 & \text{concurrent} & 2 \text{ (x4)} & 2 \text{ (x4)} & 2 \text{ (x1)} & 2 \text{ (x2)} \\
3 & \text{sequential} & 14 & 21 & 3 & 9 \\
 & \text{concurrent} & 3 \text{ (x8)} & 3 \text{ (x12)} & 3 \text{ (x1)} & 3 \text{ (x3)} \\
4 & \text{sequential} & 30 & 60 & 4 & 16 \\
 & \text{concurrent} & 4 \text{ (x16)} & 4 \text{ (x32)} & 4 \text{ (x1)} & 4 \text{ (x4)} \\
5 & \text{sequential} & 62 & 155 & 5 & 25 \\
 & \text{concurrent} & 5 \text{ (x32)} & 5 \text{ (x80)} & 5 \text{ (x1)} & 5 \text{ (x5)} \\
6 & \text{sequential} & 126 & 378 & 6 & 36 \\
 & \text{concurrent} & 6 \text{ (x64)} & 6 \text{ (x192)} & 6 \text{ (x1)} & 6 \text{ (x6)} \\
7 & \text{sequential} & 254 & 889 & 7 & 49 \\
 & \text{concurrent} & 7 \text{ (x128)} & 7 \text{ (x448)} & 7 \text{ (x1)} & 7 \text{ (x7)} \\
\end{array}
\]

Table 5.3.: The table compares op1d–counts for the relevant matrices \(M\) and \(L_{\tilde{X}}\) when applied via UpDown to linear and prewavelet grids with and without boundaries. Split level 0 is assumed in rows titled “sequential”, the maximum split level is assumed in rows titled “concurrent”. The number \(B_{\text{max}}\) of processes needed for the latter is given in parentheses, respectively.

- strike price: \(K = 1\)
- risk-free discount rate: \(r = 0.05\)
- maturity time: \(T = 1\)
- time steps with Rannacher Smoothing: \(n_{RS} = 3\)
- time step size: \(\delta t = 0.05\) (20 time steps)
- drifts \(\mu_j\), volatilities \(\sigma_j\), and correlations \(\rho_{ij}\): see Appx. C.2
- max. CG iterations: 200 in most settings, 400 in a selected few (marked in tables)

Solving the PDE in the prewavelet space leads to the accuracies given in Tab. 5.4. Only for a few selected configurations, it is solved in the space of piecewise linear hat functions with the aim to compare the runtimes of both variants (cf. Fig. 5.12). Since proper use of adaptivity is not part of this work, the following straightforward way is used to construct all grids used in simulations:

- The starting point is a truncated grid with boundaries \((\vec{c} = (7, \ldots, 7), \text{ varying initial level } n)\). As detailed in \[117\], the resulting full-grid-like refinement leads to good results.
- The initialization via the payoff function is followed by several (usually 4) rounds, in which the 25 (or less) subspaces containing the largest surpluses are refined.

---

157
5. Dimensionally Adaptive Sparse Grids in Action

The total number of added subspaces is limited to either 100 or 300. During the simulation, no refinement or coarsening takes place.

The shown results are all computed on a single node of the SNB-EP test platform. Sets of $T$ threads are used to apply 1-D operators. In total, $B$ such sets process $B' \geq B$ subtrees arising from the splitting of the FE operators’ UpDown trees. For $B' > B$, a static round-robin-like parallelization scheme is used. The goal to make best possible use of the platform’s parallel resources leads to configurations with $B \cdot T \approx 16$.

Numerical Quality and Performance

Table 5.4 presents accuracy and performance results obtained for different grid sizes for two- to seven-asset baskets. The number of DoFs (degrees of freedoms, i.e., inner grid points), absolute and relative errors obtained for the point-wise evaluation of the computed option price, and the runtime are listed for every simulation. In addition, the speedup over a single-threaded program execution ($T = B = 1$) is given in parentheses.

The errors are computed with respect to reference results from corresponding Monte Carlo simulations with $10^{10}$ paths. Comparing accuracies only (not computed prices), the results are in line with recent works about SASG that use similar problem setups for the pricing of European basket put options. For two- and three-asset baskets, about twice as many DoFs are used compared to [66], leading to relative errors that are similar or even lower. The simulation of the four-asset basket already proves difficult in [117] and [66], where the comparatively high volatilities and correlations are identified as the problem’s cause. Accordingly, the relative error obtained for SASG in these works is only slightly smaller than the error obtained here. The five-asset problem brings the SASG implementations in [117] and [66] to their limits, as the number of DoFs can not be further increased. In [117] (about 2 years ago), a hash-map-based simulation of the untransformed Black-Scholes equation with 90k DoFs takes around thirteen hours on a twelve-core workstation. In [66], a more optimized version of this simulation with 98k DoFs completes within 48 minutes on the same sixteen-core SNB-EP platform that is used for the experiments here.

For baskets with more than five assets, no results computed on SASG are available for comparison. However, results published for CT simulations of five- and six-asset baskets underline the competitive quality of the DASG simulations’ results. In [11], five-asset baskets are simulated with a total of around $10^7$ DoFs on 56 combi-grids. The results yield slightly worse relative errors compared to the DASG results, despite a homogeneous parameter configuration with equal volatilities and correlations in all dimensions. 50 time steps are performed in 216 seconds on an InfiniBand-connected cluster of 16 nodes and a total of 128 cores. For six-asset baskets, a total of $10^8$ DoFs in 84 combi-grids are used, which leads to solution times of 2025 seconds on the same cluster. The relative error is one order of magnitude lower than the one obtained for 461k DoFs on DASG, however,
### 5.3. Sparse Grids for Solving Partial Differential Equations

<table>
<thead>
<tr>
<th>$d = 2$</th>
<th>$(T = 2, B = 3)$</th>
<th>$d = 3$</th>
<th>$(T = 4, B = 4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DoFs</td>
<td>$\text{err}_{\text{abs}}$</td>
<td>$\text{err}_{\text{rel}}$</td>
<td>solution time (speedup)</td>
</tr>
<tr>
<td>5k</td>
<td>3.9e−5</td>
<td>3.2e−4</td>
<td>0.195 sec (2.48x)</td>
</tr>
<tr>
<td>8k</td>
<td>1.2e−5</td>
<td>9.5e−5</td>
<td>0.291 sec (2.41x)</td>
</tr>
<tr>
<td>12k</td>
<td>8.6e−6</td>
<td>6.8e−5</td>
<td>0.426 sec (2.36x)</td>
</tr>
<tr>
<td>16k</td>
<td>5.5e−6</td>
<td>4.4e−5</td>
<td>0.58 sec (2.40x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 4$</th>
<th>$(T = 3, B = 5)$</th>
<th>$d = 5$</th>
<th>$(T = 4, B = 3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DoFs</td>
<td>$\text{err}_{\text{abs}}$</td>
<td>$\text{err}_{\text{rel}}$</td>
<td>solution time (speedup)</td>
</tr>
<tr>
<td>6k</td>
<td>9.6e−3</td>
<td>9.3e−2</td>
<td>0.498 sec (3.73x)</td>
</tr>
<tr>
<td>12k</td>
<td>6.7e−3</td>
<td>6.5e−2</td>
<td>1.141 sec (3.86x)</td>
</tr>
<tr>
<td>28k</td>
<td>6.9e−3</td>
<td>6.7e−2</td>
<td>2.536 sec (3.84x)</td>
</tr>
<tr>
<td>60k</td>
<td>6.0e−3</td>
<td>5.9e−2</td>
<td>5.684 sec (3.88x)</td>
</tr>
<tr>
<td>114k</td>
<td>6.2e−3</td>
<td>6.1e−2</td>
<td>12.83 sec (4.01x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 6$</th>
<th>$(T = 4, B = 4)$</th>
<th>$d = 7$</th>
<th>$(T = 4, B = 4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DoFs</td>
<td>$\text{err}_{\text{abs}}$</td>
<td>$\text{err}_{\text{rel}}$</td>
<td>solution time (speedup)</td>
</tr>
<tr>
<td>12k</td>
<td>4.5e−3</td>
<td>5.2e−2</td>
<td>1.998 sec (4.84x)</td>
</tr>
<tr>
<td>57k</td>
<td>2.5e−3</td>
<td>2.8e−2</td>
<td>11.05 sec (5.04x)</td>
</tr>
<tr>
<td>120k</td>
<td>1.6e−3</td>
<td>1.8e−2</td>
<td>28.85 sec (5.31x)</td>
</tr>
<tr>
<td>270k*</td>
<td>1.2e−3</td>
<td>1.4e−2</td>
<td>92.28 sec (5.73x)</td>
</tr>
<tr>
<td>461k*</td>
<td>6.5e−4</td>
<td>7.4e−3</td>
<td>140.9 sec (4.33x)</td>
</tr>
</tbody>
</table>

Table 5.4.: Numerical and performance results for the pricing of two- to seven-asset European basket call options are shown. Asterisks in the first column mark large settings where the CG iterations are limited to 400 instead of 200.
homogeneous conditions are assumed again. In [117], another result for a six-asset CT simulation of European basket call options is given that relativizes the significance of this difference in accuracy. Although homogeneous conditions are assumed in this CT simulation, too, the comparison with the DASG simulation for 461k DoFs yields nearly identical relative errors. As for seven-asset baskets, the DASG-based solver presented here is the first sparse-grid-based solver that succeeds in this high-dimensional setting. Finally, as indicated in the table, the CG solver is allowed to run up to 400 iterations for the larger six- and seven-dimensional settings. This high number is, however, only reached during the initial smoothing phase or not at all. In later time steps, the solver typically converges after 200–250 iterations or even less.

It should be noted that speaking of convergence rates may not be adequate when looking at the errors given for different grid sizes. After all, the control over the adaptive refinement around local features (such as the payoff function’s characteristic \((d - 1)\)-dimensional kink) is quite limited with DASG. In a few cases, the accuracy at the evaluation point even decreases slightly when adding new high-level subspaces, which can be an indicator that a finer grid is needed to fully capture local features. Despite the fact that such effects occur for the problematic 4-D case\(^8\) (they are also observed in [117]), the tables show the clear tendency that investing in more DoFs improves the accuracy of the solution.

In order to judge the overall performance of the DASG simulations, it makes sense to look at the SASG simulations again for comparison. The 2-D and 3-D case are not that relevant, since highly-tuned hash-map-based implementations manage to determine an option’s fair price in 5–100 seconds on a single node as well [66]. The cases of four- and five-asset baskets are most interesting. In [66], the Black-Scholes equation is solved on SASG on up to 512 cluster nodes of the SNB-EP type. The implementation is based on the explicit distributed assembly and application of the system matrix and achieves best case runtimes of 13 seconds for 58k DoFs in 4-D and 36 seconds for 98k DoFs in 5-D. Coincidentally, the DASG implementation achieves very similar runtimes when using about twice as many DoFs in a single-node configuration: The 4-D simulation finishes in 12.8 seconds for 114k DoFs, the 5-D simulation takes 40.9 seconds to finish for 212k DoFs. These are excellent results, especially considering that the accuracies obtained by the DASG implementation are only marginally worse.

The speedups over the single-threaded configuration \((T = B = 1)\) given in parentheses in Tab. 5.4 need some explanation. Since the PDE is solved in the prewavelet space, only \(d + 1\) branches of \(UpDown\) trees are available for concurrent processing when applying the system matrix \((d\) branches for \(L_X\), 1 branch for \(M\), cf. Tab. 5.3\). The choice of \(T\) and especially \(B\) is thus often dictated by the problem, and sometimes a choice is just difficult (see 5-D and 6-D cases). In 2-D, \(T = 5\) would be the obvious choice, however, such a high number of threads introduces noticeable synchronization overhead when sharing

\(^8\) Additional unlisted simulations of the 4-D case with 1–3 million DoFs have all lead to a relative error of around 6.0e−2.
the work related to a rather small number of subspaces. Above all, the specified timings include everything after the grid and solver setup phase, which also counts in sequential tasks related to grid management. Last but not least, the fully parallel case with 16 used threads puts high pressure on the memory channels as evidenced by the strong scaling results in Sect. 4.3.1. Hence, speedups approaching 15x or 16x are unrealistic, in fact, the speedups of 7x and higher observed for the 7-D setting are already very impressive.

Finally, a runtime comparison between simulations using prewavelets and piecewise linear hat functions is shown in Fig. 5.12. For the linear hat functions, \( T \) and \( B \) are chosen according to the same criteria as before, and yet, the benefit of using prewavelets is more than visible. It should be further remarked that the numerical quality of the results obtained for the hat functions is also lower, caused by a more ill-conditioned system matrix. The huge differences in the runtimes, especially for the larger settings, are therefore partially due to slower convergence of the CG solver. E.g., although the limit is raised to 600 CG iterations for the largest simulation in 7-D, this limit is still reached in all 20 time steps. The prewavelets on the other hand require around 400 iterations in the first time steps, before the number gradually goes down to less than 200 in the last time step.

**Figure 5.12.:** The plot refers to the largest simulation listed for each dimensionality \( d \) in Tab. 5.4. The bars indicate how much faster the simulation finishes when using prewavelets instead of linear hat functions.

**Conclusions and Outlook**

The presented solutions for DASG ace the option pricing challenge, and by now, if not before, their potential is obvious. Besides achieving both impressive numerical results and solution times, the tremendous impact of the previously described algorithmic optimizations related to the UpDown scheme are proven in a complex real setting. It is further shown that the approach is able to compete with a cluster implementation of SASG in terms of accuracy and time to solution, while running only on a single node. At
the same time, the approach is much more flexible and scales even to higher dimensions, as is demonstrated by the first-time numerical pricing of seven-asset baskets via a direct sparse grid approach.

Although the implementation is already quite mature at this point, several extensions are thinkable. A more elaborate strategy for adaptive refinement might lead to a more efficient sampling of the domain. Additionally, the ideas regarding grid truncation described for the Modified CT in [117, 11] might further improve the solution’s accuracy. But of course, the flexibility of the solutions also permits their application to completely other settings of moderate dimensionality.

Finally, the presented concepts for the parallelization of *UpDown* via *UpDownScheduler* are equally applicable to distributed settings – when linked to a cost-benefit analysis that considers the communication overhead. At this time, an MPI-based implementation already exists and yields correct results on a small InfiniBand-connected cluster of SNB-EP nodes. However, the conducted performance experiments are spoiled by so far unresolved performance issues that appear with the activation of OpenMP. Hence, these results are not addressed in the discussion.  

### 5.4. Applying the Concepts to Spatially Adaptive Sparse Grids

In this application, I extend my previous considerations with regard to DASG to more general SASG. Admittedly, the following paragraphs are not completely in line with the other applications presented in this chapter, as the developed solutions are not “applied” in a particular setting. The application I discuss here is one of concepts. It resembles the procedure exercised in Chapt. 4 only now, no more self-established design principles need to be put to test. The principles’ validity is by now proven for DASG implementations, and here, I simply test the hypothesis that their application to SASG implementations is beneficial, too. The extraordinary outcome of this test is twofold:

1. For the examined case of basis transformations, the developed SASG implementation exhibits so far unseen parallel performance on shared memory systems. Experiments indicate an asymptotic complexity that is far superior in both space and time compared to that of a highly optimized hash-map-based implementation.

2. The interface design and the extension points of the software solutions for DASG (see Sect. 4.4) facilitate the development of the SASG components. Not only do

---

9 The same problems of the hybrid implementation is observed on Xeon Phi (where intra-node MPI communication takes place in shared memory), whereas the purely OpenMP-based program variant is unaffected. Note that these experiments on the Xeon Phi are only for proof of concept; the platform is only competitive for programs maintaining a high degree of parallelism at all times.
the interfaces allow for the new components’ seamless integration, the software structure itself clearly guides the process of code vectorization and promotes the reuse of existing 1-D operator implementations.

The whole section can thus be considered a part of the outlook of this thesis, except I do not stop at describing the potential of the developed concepts. I go one step further and demonstrate how these concepts can be profitably applied in another context. As for the current state of the SASG implementation, it is from my perspective still an evolving idea whose potential is all but exhausted.

Since the key-value data structure discovered by R. Jacob is explained in detail in Sect. 3.3.3 and Sect. 3.3.5, I only give a short recapitulation of the general principle here. The description of a major enhancement to the original approach follows. It allows to replace a general $O(N \log N)$ sorting step of the whole data array with an efficient $O(N)$ mapping. In its current form, the optimization works for a degenerated UpDown scheme that can be completed in a simple loop over the dimensions. With an emphasis on software interfaces, I then explain the parallelization and the vectorization scheme. Finally, I show the extremely promising outcome of the performance comparison with an implementation based on the SG++ hash map.

### 5.4.1. The Key-Value Approach Briefly Revisited

For additional details about the underlying idea of the key-value storage form refer to Sect. 3.3.3. Here, only the coarse structure shall be recapitulated. Note also that in this thesis only the data structure for the case without boundaries is treated (which also holds for the implementation). There is, however, no conceptual problem regarding its extension. Judging from the results obtained for the non-adaptive program variant, an implementation with boundaries might lead to even better results compared to a hash map implementation. This is because the data locality exhibited by the coefficient layout for the standard tasks is further improved by the introduction of points on the boundary. Figure 4.9 in Sect. 4.3.2 illustrates the coefficient layout for the boundary case and indicates the clustering of affected coefficients during evaluation. More information about this topic are found in [25].

The data structure under consideration is built around the bijective mapping of a $d$-dimensional (regular) sparse grid’s points to a dense index range. The mapping’s structure leads to an enumeration of the grid points as seen in a multi-recursive breadth first traversal of the sparse grid tree, hence the name bfs mapping. For its explicit form in 1-D and $d$-D, see (3.30) and (3.31), respectively. For regular (and truncated) sparse grids with a priori known structure, it suffices to store a linear coefficient array, as grid-specific information can be recomputed on demand. For SASG in contrast, the grid’s structure needs to be maintained explicitly. To this end, an SASG is considered an embedding of a larger regular grid, and all grid points (i.e., the coefficients) are labeled with their
bfs indices with respect to the regular grid. This leads to a list of index-coefficient pairs, as formalized in (3.32), and the designation of the idea as “key-value” approach. Finally, “enumerating” the SASG’s grid points by arranging the key-value pairs in ascending order with respect to the keys ensures that all 1-D subgrids reside compactly in memory.

The latter property is the central element of the approach, rendering it so interesting for high performance implementations. I/O-wise, the representation of an SASG as a sorted list of the described format is optimal for the application of a 1-D operator, as a single sequential scan of the data is all that it takes. Of course, the unidirectional principle postulates the application of operators with respect to all dimensions, and so the optimality can only be exploited for a second and third dimension if the list’s contents are reordered in between passes. This is done by computing all points’ level-index vector pairs via the bfs mapping’s inverse, cyclically shifting the vectors’ components, remapping the vectors to new bfs indices, and reestablishing the sorted order of the sequence. Take the example of linear hierarchization in d dimensions: The whole procedure requires $O(dN \log N)$ operations due to $d$ passes that are dominated by $O(N \log N)$ time sorting steps (if for instance quicksort or radix sort are used). In the next section, I explain how to avoid the sorting step and thus reduce the overall cost to $O(dN)$ operations.

5.4.2. Elimination of the Sorting Step

Consider the three-dimensional level 3 sparse grid in Fig. 5.13 and the linear hierarchization task. If the grid contains all 31 points and the bfs mapping index (3.31) is computed for $n = 3$, a systematic $O(N)$ permutation exists that removes the need for the sorting step between consecutive passes of the algorithm (see [78]). If gaps are introduced, e.g., by dropping the points in blue brackets in Fig. 5.13, the permutation principle fails, as it assumes a priori known subgrid sizes. The fall-back solution in these cases is a generalized sorting step exercised after the remapping of the bfs indices. However, an analysis of the dimensional shift’s effects reveals that this sorting step can be circumvented for the hierarchization task.
5.4. Applying the Concepts to Spatially Adaptive Sparse Grids

Figure 5.13.: Left: A 3-D level 3 sparse grid is shown in key-value representation. The storage order is \( xyz \), i.e., rows correspond to 1-D subgrids sampling the \( x \)-dimension. Grid points \((\vec{l}, \vec{i})\) are labeled with their keys \( bfs(1)(x, 1_x) \) according to (3.31) (with \( n = 3 \) for a dense index range). Colors indicate class membership with respect to the 1-D keys \( bfs(1)(l_x, i_x) \). Right: The grid is shown after a dimensional shift (storage order \( yzx \)), still with the old keys to visualize the effects of the permutation. Close inspection of a class’s points reveals that their relative order is unaffected by the shift. This observation is central and holds equally for regular grids and SASG.
The cyclical shift $l_j \rightarrow l_{j'}$, $j = (j' + 1) \mod d$, $j \in D$ of level vector $\vec{l}$ (and index vector $\vec{i}$, respectively) turns the "fastest" dimension into the "slowest". Consider dimension $x$ in Fig. 5.13. In the left part of the depiction, the illustrated grid's columns group grid points sharing the same 1-D bfs index $\text{bfs}^{(1)}(l_x, i_x)$. One can say, the 1-D bfs index defines classes for the grid points. In the right part, these classes form $(d-1)$-dimensional subgrids. Observe that the relative order of each class's individuals is identical in both grid illustrations. An algorithmic exploit of these circumstances is shown in Alg. 7. In a first pass over the data, the number of individuals is determined for each class, i.e., the 1-D bfs indices with respect to the currently fastest dimension (here: dimension $x$) are computed and their occurrences are counted. As indicated in the illustration, the individuals of a class form a $(d-1)$-dimensional subgrid in the shifted grid. The ascending 1-D bfs index determines the order of the subgrids, and so all subgrid offsets can directly be derived from the classes' cardinalities. Consequently, only one more pass over that data is needed to compute and apply the mapping: Each class's (subgrid's) members are already pre-sorted, and with all subgrid offsets known, the new locations in the output array are easily determined in an incremental fashion.

For the parallelization of Alg. 7 only minor modifications are necessary. When using static load balancing by splitting the input vector into equal-sized parts, the only thing that changes is the computation of the offsets. The two passes over the data are thus perfectly parallelized. This leads to a much higher parallel efficiency during the second pass compared to an implementation that relies on general sorting.
5.4. Applying the Concepts to Spatially Adaptive Sparse Grids

Algorithm 7 Procedure \textsc{prepareNextPass} maps the key-value pairs in input vector \(\vec{a}\) (length \(N\)) to output vector \(\vec{b}\), preparing the 1-D operator application in the next dimension. The procedure requires \(\mathcal{O}(N)\) operations and uses an extra array of size \(2^n - 1\) for offset computations, where \(n\) is the maximum 1-D level with respect to the current fastest dimension. Some complexity is hidden behind the call to function \textit{shiftKey} (line 16): \(bfs^{(d)}(\vec{l}, \vec{i})\) is used to obtain level and index vectors \(\vec{l}\) and \(\vec{i}\), which are then cyclically shifted, before the new bfs key is determined via \(bfs^{(d)}(\vec{l}, \vec{i})\).

1: \textbf{function} \textsc{prepareNextPass}(\(n, N, \vec{a}, \vec{b}\))
2: \quad \text{numClasses} \leftarrow 2^n - 1 \quad \triangleright \text{number of different classes (1-D bfs indices)}
3: \quad \text{offsets}[0 : \text{numClasses}] \leftarrow 0
4: \quad \triangleright \text{1st pass: count elements in classes}
5: \quad \textbf{for} \(j = 0, \ldots, N - 1\) \textbf{do}
6: \quad \quad \text{class} \leftarrow \text{extract1dBfsKey}(\vec{a}[j].\text{key}) \quad \triangleright \text{determine class (1-D bfs index)}
7: \quad \quad \text{offsets}[\text{class}] \leftarrow \text{offsets}[\text{class}] + 1 \quad \triangleright \text{increase count for class}
8: \quad \textbf{end for}
9: \quad \triangleright \text{turn counts of elements in classes into class offsets}
10: \quad \textbf{for} \(c = 1, \ldots, \text{numClasses} - 1\) \textbf{do}
11: \quad \quad \text{offsets}[c] \leftarrow \text{offsets}[c] + \text{offsets}[c - 1]
12: \quad \textbf{end for}
13: \quad \triangleright \text{2nd pass: write updated keys-value pairs to output vector } \vec{b}
14: \quad \textbf{for} \(j = 0, \ldots, N - 1\) \textbf{do}
15: \quad \quad \text{class} \leftarrow \text{extract1dBfsKey}(\vec{a}[j].\text{key})
16: \quad \quad b[\text{offsets}[\text{class}]].\text{key} \leftarrow \text{shiftKey}(\vec{a}[j].\text{key}) \quad \triangleright \text{write remapped key to } \vec{b}
17: \quad \quad b[\text{offsets}[\text{class}]].\text{value} \leftarrow a[j].\text{value} \quad \triangleright \text{copy value to } \vec{b}
18: \quad \quad \text{offsets}[\text{class}] \leftarrow \text{offsets}[\text{class}] + 1 \quad \triangleright \text{update offset for next point}
19: \quad \textbf{end for}
20: \textbf{end function}
5. Dimensionally Adaptive Sparse Grids in Action

5.4.3. Vectorization in an Adaptive Setting

There is no general answer to the question how vectorization is best introduced in fully adaptive settings. In this particular case, the design of the DASG code guides the process and leads to an elegant solution with many benefits. Same as for DASG, vectorization is best done across several 1-D trees, and it is best realized through temporary buffers that take care of coefficient alignment. Yet, an SASG can not be expected to contain many subtrees of uniform shape, and even if it does, how are they spotted and bundled efficiently?

I approach the problem similarly to how it is done frequently in sparse linear algebra. For instance, if sparse matrices are small enough, their treatment as dense matrices in highly optimized kernels (e.g., see collection of BLAS\textsuperscript{10} routines) often leads to better performance. A comparable approach in the SASG setting therefore suggests to treat any 1-D subtree as a full binary trees regardless of possibly missing nodes. This motivates a scheme, in which adaptive 1-D bfs trees are padded with dummy nodes before “dense” operators are applied to them. Of course, not all operators are fully compatible with this approach, but at least the 1-D operators for DASG grids with non-overlapping bases (the piecewise linear and polynomial bases) can be directly recycled.

In compliance with the software design described in Sect. 4.4.2, the SubspaceStackBuffer interface is implemented in a C++ class VectorizedAdaptiveStack. The buffer object and its usage are graphically described in Fig. 5.14. Before starting, the buffer’s contents are zeroed. Then, depending on the configured width $W$ of the buffer (here: the vector register width $W = W_{AVX} = 4$), several 1-D trees are fed into the buffer, and their maximum level is remembered for operator application later on. As indicated in the illustration, the buffer can facilitate the process of loading by internally mirroring the structure of a (vectorized) 1-D bfs tree. It thus elegantly avoids conditional branching in the reading (and writing back) of arbitrarily shaped 1-D trees. Note that Down operators are likely to alter the dummy nodes in the buffer, but this is not a problem as their values are simply ignored afterwards.

Last, applying the iterator principle again proves effective in hiding specialized mechanisms. Loading (and writing back) the buffer contents requires efficient extraction of 1-D trees from the key-value stream. A small trick helps to detect the 1-D roots: Performing a left shift on all bfs indices allows to use the least significant bit as flag that marks the 1-D roots. This leaves only 63 bits for the integer representation of the bfs index, which is, however, more than enough. Since the original bfs index is restored through a simple right shift, the whole mechanism barely introduces overhead and can be completely hidden behind the interface of an iterator.

\textsuperscript{10}http://www.netlib.org/blas/

\textsuperscript{11}Many sophisticated strategies can be imagined with the goal to achieve a high degree of uniformity of the processed trees. Here, the focus is only on the most basic strategy that sequentially loads one set of $W$ trees after the other.
5.4. Applying the Concepts to Spatially Adaptive Sparse Grids

Figure 5.14: From a stream of key-value pairs, the coefficients of the 1-D trees need to be extracted efficiently, in order to be temporarily aligned in the buffer structure (right) for vectorized processing. In the top left, such a stream is abstracted as a sequence of 1-D bfs indices. The buffer uses the same indexing scheme, i.e., a point from subgrid \( t_j, j \in \{0, \ldots, W-1\} \) with 1-D bfs index \( b \) is copied to buffer index \( b \cdot W + j \), \( W \) being the buffer width.

5.4.4. Results

In the following, linear hierarchization is benchmarked in a performance comparison of the new key-value data structure and the SG++ hash map. Besides program runtime, memory consumption is also made a discipline of the competition. To get an impression of how the choice of levels and dimensions influences the performance, regular sparse grids of different sizes (without boundaries) are used in the benchmarks. Note that although performance results achieved by R. Jacob’s prototypical implementation are not shown due to unavailability, the implementation presented here can be expected to achieve superior performance, as the general sorting step has meanwhile been replaced by a more efficient direct reordering of the key-value pairs.

Figure 5.15 shows the outcome of the comparison with the hash map implementation. In order to give an impression of expected asymptotic behavior, both implementations are benchmarked across a two-dimensional parameter range. The dimensionality \( d \) is varied between 2 and 10, the grid level is varied between 5 and 10. Figure 5.15a shows sequential speedups of the key-value approach over the hash map on the SNB-EP CPU platform. For small grids, the hash map is 3-4 times faster, but the tables turn with growing size of the grids, as the key-value implementation then achieves speedups of more than 5x.

The explanation is found in Fig. 5.15b. The plot shows how many times higher the peak memory allocation of the program runs related to the hash map are compared to those.
5. Dimensionally Adaptive Sparse Grids in Action

Figure 5.15.: Sequential program executions of linear hierarchization on regular sparse grids are compared for the hash map and the key-value implementation. It is shown how much longer program runs take (cf. Fig. 5.15a) and how much higher the peak memory allocation is (cf. Fig. 5.15b) when comparing the former to the latter implementation.

related to the key-value implementation. One immediately notices a strong correlation between both plots. For small grids, the hash map is more memory efficient, and it is likely to fit into the platform’s 2x20 MB L3 cache. The operator formulation relying on random access benefits from these conditions, besting the key-value version which goes to additional effort as it rearranges the data in between passes. The numbers confirm this analysis: The overall memory footprint of the key-value implementation is around $6N \cdot 8\text{ bytes}$.\textsuperscript{12} SG++ keeps only a single data copy, but the measurements show that the hash map’s rapidly growing memory consumption exceeds the L3 cache size at around 70,000 grid points. And indeed, in Fig. 5.15a the 70k-point-line would roughly run in between the break-even line (speedup 1) and the 2.4x line, strongly resembling the 2.5x line from Fig. 5.15b. As this is the point where the key-value approach really starts to dominate the setting (cf. monotonously growing speedups for larger grids), the plot certifies the key-value implementation’s high cache efficiency. Note also that the hash map allocates around 16 GB for a 10-D level 10 grid of around 32 million grid points. The key-value approach thus represents the only viable solution if even larger settings are to be tackled.

\textsuperscript{12} The value $6N \cdot 8\text{ bytes}$ is highly implementation-specific. The “sorting step” in Alg. 7 is not in place, i.e., two vectors $\vec{a}, \vec{b}$ of $(8 + 8)$-byte key-value pairs are needed. In addition, the instantiations of the GridStorage and Sweep1dInterface interfaces each store one more copy of the 8-byte keys internally. At least the latter two lists are not crucial for the algorithm’s execution, and so the memory footprint could even be lowered by about one third.
5.4. Applying the Concepts to Spatially Adaptive Sparse Grids

The last part of the performance analysis is about parallel efficiency. Because of the unavailability of a parallel version of the SG++ operation for hierarchization, only the parallel efficiency of the key-value approach is examined. In Fig. 5.16, the transform’s strong scaling behavior on the CPU NUMA system is visualized for different grid sizes. The study reveals that the overhead of parallelization outweighs the gain for small grids (cf. Fig. 5.16a). A possible explanation is the number of thread barriers per dimensional pass, which amounts to three in the current implementation. For larger grids in comparison, the parallel efficiency increases considerably (cf. Fig. 5.16b). The speedup on the CPU system’s first NUMA node’s 8 cores approaches 7x for some scenarios. For the largest test scenarios, the scaling even continues on the second NUMA node, and overall speedups of up to 13.5x are achieved for all 16 cores. This observation bears witness of high cache efficiency, as the implementation seems to be unaffected by costly accesses into the remote NUMA node’s memory.

---

13 As stated by the SG++ developers at the time of writing, the parallel implementation of the hierarchical transform was removed from the project because of low parallel efficiency. This further strengthens the position of the key-value implementation in the direct comparison of both solutions.
6. Conclusions

In this thesis, contemporary data structures and algorithms for dimensionally adaptive sparse grids (DASG) were developed and shown to meet the requirements of modern parallel computing platforms. The presented contents were arranged in such a way, that readers following the narrative are walked through the process of co-design of data structures and algorithms that determined the steps of this thesis. The contents cover the single stages in chronological order, as they are

- **Chapter 2**:
  - learning about specifics of the hardware,
  - getting familiar with the underlying theory of the problem,
- **Chapter 3**:
  - identifying constraints imposed by theory,
  - extracting data patterns and dependencies from algorithms,
  - evaluating existing implementations w.r.t. strengths & shortcomings,
- **Chapter 4**:
  - establishing a set of design principles for algorithms & software,
  - defining data structures according to these principles,
  - defining algorithms on top of the data structures,
  - testing the resulting combination under lab conditions,
- **Chapter 5**:
  - employing the new solutions in applications,
  - looking for broader applicability of the principles.

The co-design of data structures and algorithms for high performance software is a complex undertaking that consists of interdependent subtasks. The process will always be heavily influenced by the specific problem under examination, and it is therefore hard if not impossible to find a manual, listing in detail all the steps that must be taken in its course. At least the related literature research conducted for this thesis led to disappointment and revealed no instructive hints how to best tackle this ambitious task. Despite and because of these circumstances, I am confident in claiming that the generalized approach to co-design as briefly sketched above will provide an excellent starting point for future projects and can thus be considered one of the main contributions of this work. After all, it directly led to two other impressive contributions of this thesis, which are

- a DASG implementation that is currently the fastest published shared memory implementation based on the direct, hierarchical sparse grid approach, and
- an early but nevertheless fully functioning implementation of a novel SASG data
structure that already exhibits a storage efficiency, cache efficiency, and parallel efficiency so far unseen for SASG data structures on shared memory systems.

As for the performance of the presented solutions, the classical tasks on sparse grids (hierarchical transform, evaluation, UpDown) were solved in record time. For the hierarchical transform, Amdahl’s Law defines the odds, and still, excellent results were obtained. On a dual eight-core CPU NUMA platform, the DASG implementation achieved strong scaling results of up to 7x on the first socket for problems of sufficient size. For the largest problems considered, this behavior even continued on the second socket with speedups of up to 12x. The SASG implementation achieved results of the same if not better quality. Here, the parallel efficiency was even higher for the larger problems, leading to speedups of up to 13.5x on the same NUMA platform.

The results achieved for the hierarchical transforms are of immediate relevance for the UpDown algorithm as well, as one-dimensional schemes here need to be applied to the grids in the same fashion. In addition, algorithmic optimizations for the UpDown scheme were presented related to beneficial temporary basis changes. The effectiveness of these solutions was also proven in a real setting, where European basket call options were successfully priced in 2–7 dimension on a single CPU node. The results are impressive for several reasons, as 1) the successful pricing of seven-asset options via the direct sparse grid approach is to my knowledge a new record, 2) the results are competitive to those achieved by a highly optimized SASG implementation running on 512 cluster nodes, and 3) the algorithmic optimality of the solutions allows (in contrast to the SASG cluster implementation) for further scaling of the problem size.

The task of evaluation is very relevant as well, as two common sparse grid applications, classification and regression, heavily depend on its fast completion. In this work, a specifically designed combination of data structure and algorithm combined the strengths of several previous implementations, leading to speedups of up to 13.5x over the fastest predecessor on the CPU. On the GPU and the Xeon Phi, even higher speedups were on average obtained than for the CPU, peaking at a value of 17.2x observed on the Xeon Phi for one of the considered problems. The quality of the results was confirmed by a performance benchmark in an actual classification setting, where the so far fastest (SASG-based) sparse grid classifier was clearly outperformed on regular sparse grids.

More contributions of this thesis have manifested in a set of interfaces and classes that have been shown to fulfill their purpose in various respects, as the design proved to

- be well-adapted to modern many-core platforms, allowing for highly efficient implementations of algorithms on three different platforms,
- relieve programmers to a certain extent from the need to explicitly account for a computer’s performance critical resources (e.g., vector registers, multi-threading, fast caches),
- facilitate the rapid and straightforward development of additional complex func-
tionality (such as mathematical grid operators),

- be flexible and extensible, permitting the seamless integration of both high performance legacy software (see implementations of regular sparse grids from \[24\]) and fast new functionality for SASG.

Considering planned and possible future projects, the presented software solutions are at the top of the to-do list, as they are intended to be merged into the widely used SG++ project in the near future. This also entails a planned refactoring of the SG++ project’s software interfaces and (possibly) some of its components.

Related to these plans, it will be interesting to see whether a more refined version of the SASG data structure first proposed in this thesis has the potential of breaking the long-lasting predominance of the hash map data structure. The work on this project has at this point already begun, as a variant is being developed that supports explicit boundary representations. A so far unanswered question is whether the UpDown scheme can also take advantage of the efficient $O(N)$ mapping that replaces the generalized $O(N \cdot \log N)$ sorting step in between computational phases of the hierarchical transform.

For the DASG implementation, at least three future projects are directly motivated by this thesis. The more specialized, application-related one is the final integration of the GPU-based hierarchical transform into the computational steering application described in Sect. 5.2. A more generally applicable extension to the DASG code is the distributed computation of FE operators based on the solutions described in Sect. 5.3. As already detailed there, a working MPI implementation with unresolved performance issues related to the use of OpenMP already exists, but even for a fully working version a cost-benefit analysis considering the cost of communication first needs to be done. Finally, the tool “dimensional adaptivity” must be further explored and shaped, in order to release the full potential of the solutions developed in this thesis.

Beyond these obvious options for future projects, who knows, we can also just follow the example of Monty Python and try the new solutions “for something completely different”.

175
A. Formulations of Selected 1-D Operators

A.1. Linear Hierarchization

In Lst. 3.1, the C-like stencil formulations of linear hierarchization and dehierarchization are shown. Their conciseness is a strong indicator of the programming interfaces’ usability.

Listing A.1: The stencil formulation of the linear hierarchization operator forward is exactly the same as in pseudocode (cf. Alg. 1). Its inverse backward is obtained by systematically reversing the order of all operations on the subspace stack buffer and inverting their effect. Functions left and right are those for the left and right hierarchical neighbor (cf. (3.3) and (3.4)).

```c
void forward(int n, SubspaceStackBuffer* buffer) {
    for (int ℓ = n; ℓ > 1; --ℓ) { // bottom up
        for (int i = 1; i <= 2ℓ−3; i += 2) {
            (ℓ′, i) = right(ℓ, i)
            buffer->xpay(ℓ, i, -0.5, ℓ', i');
            buffer->xpay(ℓ, i+2, -0.5, ℓ', i');
        }
    }
}

void backward(int n, SubspaceStackBuffer* buffer) {
    for (int ℓ = 2; ℓ <= n; ++ℓ) { // top down
        for (int i = 2ℓ−3; i >= 1; i -= 2) {
            (ℓ′, i) = right(ℓ, i)
            buffer->xpay(ℓ, i+2, +0.5, ℓ', i');
            buffer->xpay(ℓ, i, +0.5, ℓ', i');
        }
    }
}
```

177
A. Formulations of Selected 1-D Operators

A.2. The Prewavelet Transform

In Sect. 4.3.1 the implementation of the prewavelet transform in the virtual function \textit{forward} of the abstract \texttt{Operator1d} class was shown in Lst. 4.3. In Lst. A.2, the picture is completed with the definition of the inverse transform in form of the complementary function \textit{backward}. 


A.2. The Prewavelet Transform

Listing A.2: Directly implementing (3.13) for the prewavelets’ inverse transformation would require additional temporary values. Clever combinations of the \( t_{\ell,i} \) can, however, avoid this problem and reduce the total number of operations.

```c
void backward(int n, SubspaceStackBuffer* buffer) {
    // bottom up traversal of the hierarchy
    for (int \( \ell = n; \) \( \ell > 1; \) --\( \ell \)) {
        // goal: \( u_{\ell,i} := 1.6u_{\ell,i} + 0.4u_{\ell,i+2} + t_{\ell+1,2i} - 0.5t_{\ell+1,2i+2} \)
        int \( i_{\text{max}} = 2^\ell - 1; \)
        for (int \( i = 1; i <= i_{\text{max}}; i += 2 \)) {
            buffer->scale(\( \ell, i, 0.8 \));
        }
        for (int \( i = 2; i < i_{\text{max}}; i += 2 \)) {
            buffer->set(\( \ell, i, \ell, i-1 \));
            buffer->xpay(\( \ell, i, 1.0, \ell, i+1 \));
        }
        // now have: \( t_{\ell,i} := 0.8u_{\ell+i+1} \)
        for (int \( i = 2; i < i_{\text{max}}; i += 2 \)) {
            buffer->xpay(\( \ell, i-1, 0.5, \ell, i \));
            buffer->xpay(\( \ell, i+1, 0.5, \ell, i \));
        }
        // now have: \( u_{\ell,i} := 1.6u_{\ell,i} + 0.4u_{\ell,i+2} \)
        if (\( \ell < n \)) { // resolve redundancy in \( t_{\ell+1,i} \), compute \( t_{\ell,i} \)
            buffer->xpay(\( \ell, 1, 1.0, \ell+1, 2 \));
            buffer->xpay(\( \ell, 1, -0.5, \ell+1, 4 \));
            for (int \( i = 3; i < i_{\text{max}}; i += 2 \)) {
                buffer->xpay(\( \ell, i, -0.5, \ell+1, 2i-2 \));
                buffer->xpay(\( \ell, i, 1.0, \ell+1, 2i \));
                buffer->xpay(\( \ell, i, -0.5, \ell+1, 2i+2 \));
            }
            buffer->xpay(\( \ell, i_{\text{max}}, -0.5, \ell+1, 2i_{\text{max}}-2 \));
            buffer->xpay(\( \ell, i_{\text{max}}, 1.0, \ell+1, 2i_{\text{max}} \));
            for (int \( i = 2; i < i_{\text{max}}; i += 2 \)) {
                buffer->axpy(-0.75, \( \ell, i, \ell+1, 2i \));
            }
        }
        else { // only compute \( t_{\ell,i} \)
            for (int \( i = 2; i < i_{\text{max}}; i += 2 \)) {
                buffer->scale(\( \ell, i, -0.75 \));
            }
        }
    }
    if (\( n > 1 \)) { // resolve redundancy on level 1
        buffer->xpay(1, 1, 1.0, 2, 2);
    }
}
```
B. Hierarchical Transforms for Out-of-Core Visualization

B.1. Construction of Orthonormal Wavelets

Orthonormal wavelet bases are defined by a multi-resolution analysis or multi-resolution approximation (MRA), a term coined by Mallat [86]. An MRA for $L_2(\mathbb{R})$ is formed by a nested sequence of closed subspaces $V_l \subset L_2(\mathbb{R}), l \in \mathbb{Z}$ with the following properties:

\begin{align}
V_l &\subset V_{l+1} \quad \forall l \in \mathbb{Z}, \quad \text{(B.1a)} \\
\bigcup_{l \in \mathbb{Z}} V_l &\text{ is dense in } L_2(\mathbb{R}), \quad \text{(B.1b)} \\
\bigcap_{l \in \mathbb{Z}} V_l &\equiv \{ 0 \}, \quad \text{(B.1c)} \\
f(x) &\in V_0 \Leftrightarrow f(2^{-l}x) \in V_l, \quad \text{(B.1d)} \\
f(x) &\in V_0 \Rightarrow f(x - 2^{-l}k) \in V_l \quad \forall k \in \mathbb{Z}. \quad \text{(B.1e)}
\end{align}

The theory of multi-resolution analysis states the existence of a unique function $\phi$ that satisfies a so-called scaling equation (also dilation equation) of the form

$$\phi(x) = \sum_{k \in \mathbb{Z}} c_k \cdot \phi(2x - k). \quad \text{(B.2)}$$

Through scaling and translation, orthonormal bases for the $V_l$ can be constructed from $\phi$ as

$$V_l = \text{span}\{ \phi_{l,k} \mid k \in \mathbb{Z} \} \quad \text{where} \quad \phi_{l,k}(x) := \phi(2^l x - k) \quad \text{with} \quad <\phi_{l,j}, \phi_{l,k}> = \delta_{j,k} \text{ for } l, j, k \in \mathbb{Z}. \quad \text{(B.3a,b)}$$

Furthermore, a dual hierarchy of orthogonal complement spaces $W_l$ exists satisfying

$$V_l = V_{l-1} \oplus W_{l-1} \quad \text{with} \quad <f, g> = 0 \quad \text{for } f \in V_l, g \in W_l. \quad \text{(B.4, 5)}$$

For the construction of bases for the complements $W_l$, a mother wavelet $\psi$ is derived from the dilation equation (B.2) as

$$\psi(x) = \sum_{k \in \mathbb{Z}} (-1)^k \cdot c_k \cdot \phi(2x - k). \quad \text{(B.6)}$$
Copying the steps from before, orthonormal basis functions for the $W_l$ are constructed as scaled and translated versions of $\psi$ according to

$$W_l = \text{span}\{ \psi_{l,k} \mid k \in \mathbb{Z} \} \quad \text{where} \quad (B.7a)$$

$$\psi_{l,k}(x) := \psi(2^l x - k) \quad \text{with} \quad <\psi_{l,j}, \psi_{l,k}> = \delta_{j,k} \text{ for } l, j, k \in \mathbb{Z}. \quad (B.7b)$$

Note that the construction (B.6) of the mother wavelet together with the conditions of the MRA (B.1) ensure the postulated orthogonality (B.5) of spaces $V_l$ and $W_l$.

Finally, a noteworthy property of orthonormal wavelets is that each of the four filters (low- and high-pass filters for analysis and synthesis, respectively) defines the other three through the set of filter coefficients. For the low- and high-pass filter of the analysis (father and mother wavelet), the relation is shown in (B.6).

### B.2. The Multi-Level Subspace Splitting for Reconstruction of Image Data

As detailed in Sect. 5.2.3, the hierarchical splittings based on classical sparse grid levels and on $\sqrt{h}$ grids have been employed before for visualizing data on sparse grids. In Sect. 5.2.4, the splittings are put to test in the approximation of two “unpleasant” analytic test functions. In the experiment, the $L_2$ error serves as indicator for the approximation quality. This makes sense for simulation output which often yields a certain smoothness.

From real world data this can, however, not be expected. Hence, the $L_2$ error is an unsuitable metric when it comes to restoration of data with sharp edges and high frequencies, as for example encountered in photographs. In image data compression, metrics operating on the final rasterized image are therefore used, e.g., the Mean-Squared Error (MSE) and the Peak Signal-to-Noise Ratio (PSNR) [51]. Here, we are only driven by curiosity and have no ambition of running a detailed analysis. It is left to the reader to judge how well the reconstruction of the famous “Lena” image works after decomposition with the two hierarchical splittings under consideration.

The image sequence shown in Fig. B.1 documents the reconstruction process after applying the level-wise splitting to the $513 \times 513$ image. In Fig. B.2 the $\sqrt{h}$ splitting is examined for the same image. The linear basis subsamples the image and consequently introduces star-shaped artifacts near sharp edges for both splittings. Prewavelets manage to cope with this problem better, as the low-level approximations do not interpolate. Instead, averages are formed that incorporate also knowledge about high frequencies in the input signal.

---

1 “Lena” shows the model Lena Söderberg posing as centerfold in a 1972 issue of the playboy magazine. A cropped version of the image has gained popularity in the visualization community and is used in many benchmarks related to data compression.
B.2. The Multi-Level Subspace Splitting for Reconstruction of Image Data

level 1–9, 2.3% of the data

level 1–11, 6.6% of the data

level 1–12, 11.2% of the data

level 1–13, 19.1% of the data

level 1–14, 31.5% of the data

level 1–14, 50.1% of the data

level 1–16, 75.1% of the data

level 1–17, 100% of the data

Figure B.1.: The “Lena” image is stored on a truncated sparse grid (truncation vector $\vec{c} = (9,9)$) which is first decomposed and then reconstructed using the classical level-wise sparse grid splitting. In two sequences arranged for direct comparison, the image is approximated with linear and prewavelet basis functions. Above each pair of images, the sparse grid levels and the relative amount of data used for the reconstruction are given.
Figure B.2.: The “Lena” image is stored on a truncated sparse grid (truncation vector $\vec{c} = (9, 9)$) which is first decomposed and then reconstructed using the $\sqrt{h}$ splitting. In two sequences arranged for direct comparison, the image is approximated with linear and prewavelet basis functions. Above each pair of images, the “levels” (cf. blue dotted lines in Fig. 5.6b) and the relative amount of data used for the reconstruction are given.
C. Solution of the Black-Scholes Equation for Option Pricing

C.1. Transformation into the Heat Equation

The parabolic multi-variate Black-Scholes equation

\[ \frac{V}{\partial t} + \frac{1}{2} \sum_{i,j=0}^{d-1} \sigma_i \sigma_j \rho_{ij} S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=0}^{d-1} \mu_i S_i \frac{\partial V}{\partial S_i} - rV = 0 \quad (C.1) \]

can be transformed into the easier-to-solve heat equation in five steps. We more or less copy these steps from [117] and include them here for convenience and completeness.

1.) An initial value problem is obtained through the time-reverting substitution \( \tau := T - t \) and \( \frac{\partial V}{\partial t} := -\frac{\partial V}{\partial \tau} \):

\[ \frac{V}{\partial \tau} - \frac{1}{2} \sum_{i,j=0}^{d-1} \sigma_i \sigma_j \rho_{ij} S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} - \sum_{i=0}^{d-1} \mu_i S_i \frac{\partial V}{\partial S_i} + rV = 0 \quad (C.2) \]

2.) The variable coefficients are removed by switching to logarithmic coordinates \( x_i := \log S_i \). The chain rule gives

\[ \frac{\partial V}{\partial S_i} = \frac{1}{S_i} \cdot \frac{\partial V}{\partial x_i}, \quad \frac{\partial^2 V}{\partial S_i \partial S_j} = \frac{1}{S_i S_j} \cdot \frac{\partial^2 V}{\partial x_i \partial x_j}, \quad \frac{\partial^2 V}{\partial x_i^2} = \frac{1}{S_i^2} \cdot \frac{\partial^2 V}{\partial x_i^2} - \frac{1}{S_i^2} \cdot \frac{\partial V}{\partial x_i} \]

and plugging into (C.2) for \( i \neq j \) results in:

\[ \frac{V}{\partial \tau} - \frac{1}{2} \sum_{i,j=0}^{d-1} \sigma_i \sigma_j \rho_{ij} \frac{\partial^2 V}{\partial x_i \partial x_j} - \sum_{i=0}^{d-1} \left( \mu_i - \frac{1}{2} \sigma_i^2 \right) \frac{\partial V}{\partial x_i} + rV = 0 \quad (C.3) \]

3.) An eigendecomposition of the symmetric covariance matrix \( \Sigma := (\sigma_i \sigma_j \rho_{ij})_{i,j} \in \mathbb{R}^{d \times d} \) into real eigenvalues \( \lambda_i \) and orthogonal eigenvectors \( \vec{q}_i \) leads to the transformation matrix \( Q := (\vec{q}_0, \ldots, \vec{q}_{d-1}) \) of the principal axis transformation (PAT) \( \vec{y} := Q^T \vec{x} \), which eliminates the correlations \( \rho_{ij} \). The chain rule gives

\[ \frac{\partial V}{\partial x_i} := \sum_{j=0}^{d-1} q_{ij} \frac{\partial V}{\partial y_i}, \quad \sum_{i,j=0}^{d-1} \sigma_i \sigma_j \rho_{ij} \frac{\partial^2 V}{\partial x_i \partial x_j} := \sum_{i=0}^{d-1} \lambda_i \frac{\partial^2 V}{\partial y_i^2} \]
C. Solution of the Black-Scholes Equation for Option Pricing

from which follows for (C.3):

\[ \frac{\partial V}{\partial \tau} - \frac{1}{2} \sum_{i,j=0}^{d-1} \lambda_i \frac{\partial^2 V}{\partial y_i^2} - \sum_{j=0}^{d-1} \left( \sum_{i=0}^{d-1} \left( \mu_i - \frac{1}{2} \sigma_i^2 \right) p_{ij} \right) \frac{\partial V}{\partial y_i} + rV = 0 \]  
(C.4)

4.) A linear translation \( z_i := y_i + \tau \cdot \hat{\mu}_i \) leads to cancellation of the drift term, as the substitution of \( \frac{\partial V(y, \tau)}{\partial \tau} := \hat{\mu}_i \frac{\partial V(z, \tau)}{\partial z_i} + \frac{\partial V(z, \tau)}{\partial \tau} \) in (C.4) gives:

\[ \frac{\partial V}{\partial \tau} - \frac{1}{2} \sum_{i,j=0}^{d-1} \lambda_i \frac{\partial^2 V}{\partial z_i^2} + rV = 0 \]  
(C.5)

5.) The reactive term is then removed through the substitution \( V := u \cdot e^{-r\tau} \), as the derivatives are given as

\[ \frac{\partial V}{\tau} := \frac{\partial u}{\partial \tau} e^{-r\tau}, \quad \frac{\partial^2 V}{\partial z_i^2} := \frac{\partial^2 u}{\partial z_i^2} e^{-r\tau}, \]

finally leading to the heat equation:

\[ \frac{\partial u}{\partial t} - \frac{1}{2} \sum_{i=0}^{d-1} \lambda_i \frac{\partial^2 u}{\partial z_i^2} = 0 \]  
(C.6)

Altogether, the payoff now has the form

\[ V(\vec{S}, t) = e^{-r\tau} \cdot u(\vec{z}, \tau), \]  
(C.7)

and with \( \hat{\mu}_i = \sum_{j=0}^{d-1} \left( \mu_j - \frac{1}{2} \sigma_j^2 \right) q_{ij} \), the overall transformations are given by

\[ z_i = \left( \sum_{j=0}^{d-1} q_{ij} \log S_j \right) + (T - t) \cdot \hat{\mu}_i, \]  
(C.8)

\[ S_i = \exp \left( \sum_{j=0}^{d-1} q_{ij} (z_j - (T - t) \cdot \hat{\mu}_i) \right). \]  
(C.9)

C.2. Documentation of Simulation Parameters

All simulations of the Black-Scholes model for the pricing of European call options were run with a constant time step size \( \delta \tau = 0.05 \) in the interval \( \tau \in [0, T = 1] \). The risk-free
discount rate was also chosen equal in all simulations as \( r = 0.05 \). The drift vector \( \vec{\mu} \), the volatilities \( \vec{\sigma} \), and the correlation matrix \( P \) (with entries \( \rho_{ij} \)) varied with the size \( d \) of the different simulated baskets and are given in the following.

- **\( d = 2 \)**:
  \[
  \vec{\mu} := \begin{pmatrix} 0.08 \\ 0.09 \end{pmatrix}, \quad \vec{\sigma} := \begin{pmatrix} 0.3 \\ 0.4 \end{pmatrix}, \quad P := \begin{pmatrix} 1.0 & -0.5 \\ -0.5 & 1.0 \end{pmatrix}
  \]

- **\( d = 3 \)**:
  \[
  \vec{\mu} := \begin{pmatrix} 0.1 \\ 0.02 \\ 0.04 \end{pmatrix}, \quad \vec{\sigma} := \begin{pmatrix} 0.2 \\ 0.3 \\ 0.4 \end{pmatrix}, \quad P := \begin{pmatrix} 1.0 & -0.7 & -0.1 \\ -0.7 & 1.0 & 0.1 \\ -0.1 & 0.1 & 1.0 \end{pmatrix}
  \]

- **\( d = 4 \)**:
  \[
  \vec{\mu} := \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \end{pmatrix}, \quad \vec{\sigma} := \begin{pmatrix} 0.4 \\ 0.25 \\ 0.3 \\ 0.4 \end{pmatrix}, \quad P := \begin{pmatrix} 1.0 & 0.1 & -0.4 & 0.2 \\ 0.1 & 1.0 & 0.3 & -0.1 \\ -0.4 & 0.3 & 1.0 & 0.0 \\ 0.2 & -0.1 & 0.0 & 1.0 & -0.7 \end{pmatrix}
  \]

- **\( d = 5 \)**:
  \[
  \vec{\mu} := \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \end{pmatrix}, \quad \vec{\sigma} := \begin{pmatrix} 0.4 \\ 0.25 \\ 0.3 \\ 0.4 \\ 0.35 \end{pmatrix}, \quad P := \begin{pmatrix} 1.0 & 0.1 & -0.4 & 0.2 & 0.1 \\ 0.1 & 1.0 & 0.3 & -0.1 & 0.0 \\ -0.4 & 0.3 & 1.0 & 0.0 & 0.2 \\ 0.2 & -0.1 & 0.0 & 1.0 & -0.7 \\ 0.1 & 0.0 & 0.2 & -0.7 & 1.0 \end{pmatrix}
  \]

- **\( d = 6 \)**:
  \[
  \vec{\mu} := \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \end{pmatrix}, \quad \vec{\sigma} := \begin{pmatrix} 0.4 \\ 0.25 \\ 0.3 \\ 0.4 \\ 0.35 \\ 0.4 \end{pmatrix}, \quad P := \begin{pmatrix} 1.0 & 0.1 & -0.4 & 0.2 & 0.1 & 0.3 \\ 0.1 & 1.0 & 0.3 & -0.1 & 0.0 & 0.0 \\ -0.4 & 0.3 & 1.0 & 0.0 & 0.2 & 0.1 \\ 0.2 & -0.1 & 0.0 & 1.0 & -0.7 & -0.3 \\ 0.1 & 0.0 & 0.2 & -0.7 & 1.0 & 0.5 \\ 0.3 & 0.0 & 0.1 & -0.3 & 0.5 & 1.0 \end{pmatrix}
  \]

- **\( d = 7 \)**:
  \[
  \vec{\mu} := \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \end{pmatrix}, \quad \vec{\sigma} := \begin{pmatrix} 0.4 \\ 0.25 \\ 0.3 \\ 0.4 \\ 0.35 \\ 0.4 \\ 0.25 \end{pmatrix}, \quad P := \begin{pmatrix} 1.0 & 0.1 & -0.4 & 0.2 & 0.1 & 0.3 & -0.1 \\ 0.1 & 1.0 & 0.3 & -0.1 & 0.0 & 0.0 & 0.1 \\ -0.4 & 0.3 & 1.0 & 0.0 & 0.2 & 0.1 & 0.2 \\ 0.2 & -0.1 & 0.0 & 1.0 & -0.7 & -0.3 & 0.0 \\ 0.1 & 0.0 & 0.2 & -0.7 & 1.0 & 0.5 & 0.0 \\ 0.3 & 0.0 & 0.1 & -0.3 & 0.5 & 1.0 & 0.4 \\ -0.1 & 0.1 & 0.2 & 0.0 & 0.0 & 0.4 & 1.0 \end{pmatrix}
  \]
Bibliography


[75] Intel. Intel Xeon Processor E5-100/E5-2600/E5-4600 Product Families, may 2012.


may 2006.


