COMMUNICATION EFFICIENT ALGORITHMS FOR NUMERICAL PROBLEMS ON FULL AND SPARSE GRIDS

PHILIPP HUPP

DISS. ETH NO. 22206

2014
COMMUNICATION EFFICIENT ALGORITHMS
FOR NUMERICAL PROBLEMS ON
FULL AND SPARSE GRIDS

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
PHILIPP HUPP
Dipl.-Math. Univ., Technische Universität München, Munich
MS in Mathematics, Georgia Institute of Technology, Atlanta
born 16.06.1987
citizen of Germany

accepted on the recommendation of
Prof. Dr. Peter Widmayer, examiner
Riko Jacob, PhD, co-examiner
Prof. Dr. Ulrich Meyer, co-examiner

2014
ABSTRACT

As the gap between peak performance and bandwidth on modern computers is already large and increasing further, the design of communication-efficient algorithms becomes more and more important. This effect is reinforced further, as the problems that we solve also continuously grow in size. The problems considered in this thesis stem from the field of solvers for partial differential equations (PDEs). For low-dimensional problems, stencil computations as they appear in finite difference methods are studied. For high-dimensional problems, this thesis focuses on the sparse grid combination technique, an hierarchical discretization scheme that reduces the “curse of dimensionality”. Based on extrapolation, the combination technique additionally reduces the need for global communication. For time dependent problems, e. g., time dependent PDEs, each time step is computed on several independent and much smaller discretizations, so called component grids, and the global communication shrinks to a reduce/broadcast step in between. This work analyzes this remaining communication bottleneck of the combination technique as well as hierarchization, one of the fundamental sparse grid algorithms.

In this work, theoretical considerations are used to identify the most important communication aspects of the examined problems. Communication, in this thesis, refers to both data transfer through the memory hierarchy as well as communication via message passing. For all considered problems, novel algorithmic approaches are derived and it is shown that these approaches are, with respect to communication, either optimal or within a small constant factor of the optimum. In two cases, the theoretical insights also immediately lead to efficient implementations of the respective algorithms. In detail, the contributions of this work are the following:

STENCIL COMPUTATIONS. To solve problems numerically, the solution space is first discretized, e. g., by being mapped to a full grid space, and then the numerical problem is solved in this discretized space. For PDEs this happens, for example, using stencil computations on grids. Stencil computations update the values of the grid points according to a stencil. The stencil describes which neighbors of a grid point are used for this update. This work derives new algorithms for stencil computations and proposes data layouts that exactly match the access patterns of these algorithms. The improved lower bounds are proven by carefully applying an isoperimetric result to the rounds into which an arbitrary algorithm can be split. In combination, the existing gaps between lower
and upper bounds reduce significantly, and the new bounds match for the 2-dimensional case.

**Unidirectional Hierarchization.** Hierarchization describes the base change from the full grid basis to the hierarchical basis of sparse grids. As such, hierarchization is one of the fundamental sparse grid algorithms, and furthermore it is an important preprocessing step for the communication schemes discussed later. The unidirectional principle is the dominating design pattern for sparse grids. It splits the $d$-dimensional problem into $d$ distinct phases and solves each phase working on 1-dimensional subproblems only. This work describes an implementation of the unidirectional hierarchization algorithm for the component grids of the combination technique that runs within a factor of 1.5 of the lower bound given by the unidirectional principle and the memory bandwidth of the system used.

**Divide and Conquer Hierarchization.** The unidirectional principle guides the development of many sparse grid algorithms as it simplifies their computations. It is, however, a bad choice with respect to data transfer. Due to the $d$ phases of the unidirectional principle, any unidirectional algorithm has to load each grid point at least $d$ times. This work develops an alternative approach for isotropic component grids: a divide and conquer algorithm that avoids the unidirectional principle globally but applies it recursively to smaller subproblems. Let $M$ denote the size of the internal memory and let $B$ denote the cache line size. The derived algorithm is cache-oblivious and, assuming a tall cache of size $M \in \omega (B^d)$, optimal, as it brings the leading term of the cache misses down to the compulsory scanning complexity. Furthermore, the algorithm is complemented with a non-trivial lower bound for the leading term of the non-compulsory cache misses.

**Communication Schemes for the Combination Technique.** The combination technique breaks the global communication requirements of conventional discretization approaches by splitting the problem into subproblems that can be solved independently. Global synchronization, however, is still necessary but shrinks to a reduce/broadcast step. This work presents the first systematic study of this remaining communication bottleneck and derives two communication schemes to solve it. One of the schemes exploits the hierarchical structure of sparse grids. The communication schemes are evaluated experimentally on two supercomputers, demonstrating their performance for different scenarios. Furthermore, the schemes are analyzed and a theoretical model is built to predict their performance. The model is validated using the experiments and then applied to predict scenarios that are as yet out of scope on current supercomputers due to their high computational demands.
ZUSAMMENFASSUNG


In dieser Arbeit werden theoretische Überlegungen genutzt, um die wichtigsten Kommunikationsaspekte der untersuchten Probleme zu identifizieren. Sowohl der Datentransfer durch die Speicherhierarchie als auch der Nachrichtenaustausch auf verteilten Systemen wird in dieser Abhandlung unter dem Begriff “Kommunikation” zusammengefasst. Für alle untersuchten Probleme werden neue algorithmische Konzepte hergeleitet. Zudem wird bewiesen, dass diese Konzepte in Bezug auf die Kommunikation entweder optimal oder nur einen kleinen konstanten Faktor vom Optimum entfernt sind. In zwei Fällen führen die theoretischen Einsichten darüber hinaus direkt zu effizienten Implementierungen der entsprechenden Algorithmen. Die Beiträge dieser Arbeit sind im Detail:

**Stencil-Berechnungen:** Um Probleme numerisch zu lösen wird der Lösungsraum zunächst diskretisiert. Dies passiert beispielsweise indem der Lösungsraum auf einen Vollgitterraum abgebildet wird. Das numerische Problem kann dann anschließend in diesem diskretisierten Raum gelöst werden. Bei PDGs kann die numerischen Lösung des Pro-


eine nicht triviale untere Schranke für den führenden Term der nicht obligatorischen Cache Ladevorgänge ergänzt.

**Kommunikations schemata für die Kombinationstechnik:**

This thesis is based on the following publications, which are included in parts or in an extended version:


This thesis is further based on the following manuscripts which are currently in preparation:


ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my supervisor, Riko Jacob, PhD, who always had time to discuss open problems and their next twist. I am very thankful for the freedom you granted me while offering guidance and support whenever I needed them.

I am also very grateful to Prof. Peter Widmayer for the possibility to work in his group and, of course, for examining this thesis. I also would like express my gratitude to Prof. Ulrich Meyer for examining this thesis.

In addition, I would like to thank Prof. Markus Hegland for introducing me to the communication task of the combination technique and for hosting me in Canberra, Australia. Furthermore, my deep thanks go to Jun.-Prof. Dirk Pflüger and Mario Heene who continued to address this problem with me. This project has continued for more than two-and-a-half years and has developed far beyond the initial idea. Thanks to all of you for staying in touch with telephone calls and video conferences spanning half the globe and arranging personal visits whenever possible. It was always fun to work with you.

I also would like to express my gratitude to Prof. Markus Püschel and Georg Ofenbeck during whose lecture I started the work on the implementation of the unidirectional hierarchization algorithm. In addition, I would like to thank Gerrit Buse who not only granted me access to StructuredSG [BJPM12] but also helped me to get it running. Also, I am grateful to Moritz Baumann from the IT support group for providing assistance with setting up the experiments for this project.

Furthermore, I would like to express my gratitude to the whole group of Prof. Peter Widmayer at ETH Zürich as well as to the whole group of Prof. Ernst W. Mayr at TU München. The research leading to this work was started in the latter group and finished in the former. In particular, thank you Tobias Lieber for sharing the office, being a source of knowledge and for the various geometry sessions. In addition, I would like to thank Dennis Komm and Warren Cabral for proofreading parts of this thesis. My gratitude also goes to Gero Greiner and Marcel Schöngens.

In addition, I would like to thank the Stiftung der deutschen Wirtschaft (sdw) for a scholarship during the first years of this research.

Last but not least, thank you mum and dad for being there whenever I needed your support. Also, thank you Julia for your continuous support. The next PhD is yours!
7.4.1 AllReduce 168
7.4.2 Algorithmic Notation 170
7.4.3 Lower Bounds 171
7.4.4 Sparse Grid Reduce 171
7.4.5 Subspace Reduce 172
7.4.6 Parallel Subspace Reduce 173
7.4.7 Treating Generalizations of Sparse Grids 176
7.5 Runtime Analysis 179
  7.5.1 Precise Communication-Time Formulas 179
  7.5.2 Simplifying the Formulas 181
7.6 Testbed and Experimental Setup 183
  7.6.1 Implementation and Setup of Experiments 183
  7.6.2 Exact Modeling of the Communication Times 186
  7.6.3 Systems used for Measurements 187
  7.6.4 Additional Systems Employed to Predict Runtimes 189
7.7 Simulations and Experiments 189
  7.7.1 Predicting Runtimes 190
  7.7.2 Results on Hermit 197
  7.7.3 Results on SuperMUC 203
  7.7.4 Extending the Scope to Higher Levels 205
7.8 Conclusions and Future Work 206

8 Conclusions 209

Bibliography 215
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The memory hierarchy</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Computation graph of the 1-star stencil</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Data dependencies on a sparse grid and on a full grid</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>The sparse grid combination technique</td>
<td>9</td>
</tr>
<tr>
<td>1.5</td>
<td>The sparse grid combination technique for a time-dependent problem</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>External memory model and parallel external memory model</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Number of work bands for the Diagonal Band Algorithm</td>
<td>60</td>
</tr>
<tr>
<td>3.2</td>
<td>The Row Algorithm and the Column Algorithm</td>
<td>67</td>
</tr>
<tr>
<td>3.3</td>
<td>Hypercube Band Algorithm: evaluation bands and k-intersections</td>
<td>68</td>
</tr>
<tr>
<td>3.4</td>
<td>The Hypercube Band Algorithm and the Diagonal Band Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>3.5</td>
<td>Adjacent $\ell^1$ balls lead to the Diagonal Band Algorithm</td>
<td>75</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation bands of the Diamond Band Algorithm</td>
<td>79</td>
</tr>
<tr>
<td>3.7</td>
<td>Sweep shape, evaluation bands and projected stencil for the Hexagonal Band Algorithm</td>
<td>81</td>
</tr>
<tr>
<td>3.8</td>
<td>Evaluation bands of the Hexagonal Band Algorithm ($s = 1$)</td>
<td>82</td>
</tr>
<tr>
<td>3.9</td>
<td>Evaluation bands of the Hexagonal Band Algorithm ($s = 2$)</td>
<td>83</td>
</tr>
<tr>
<td>4.1</td>
<td>The full grid nodal basis and the hierarchical basis</td>
<td>95</td>
</tr>
<tr>
<td>4.2</td>
<td>(De-)hierarchization as pre- and postprocessing step for the reduce/broadcast step</td>
<td>96</td>
</tr>
<tr>
<td>4.3</td>
<td>Different kinds of boundary treatment</td>
<td>107</td>
</tr>
<tr>
<td>4.4</td>
<td>A 1-dimensional grid (i.e. pole) and the respective hierarchical predecessor DAG</td>
<td>109</td>
</tr>
<tr>
<td>5.1</td>
<td>Strong scaling experiments</td>
<td>125</td>
</tr>
<tr>
<td>5.2</td>
<td>Experiments varying the size of basic blocks</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>Experiments increasing the level for different dimensions</td>
<td>127</td>
</tr>
<tr>
<td>5.4</td>
<td>Experiments comparing isotropic to anisotropic grids</td>
<td>128</td>
</tr>
<tr>
<td>5.5</td>
<td>Benchmark of combiHier with StructuredSG</td>
<td>129</td>
</tr>
</tbody>
</table>
Figure 6.1  Orthant subgrids in 1 dimension.  134
Figure 6.2  Recursive splitting of a grid into its subgrids.  137
Figure 7.1  *Sparse Grid Reduce* and *Subspace Reduce*.  164
Figure 7.2  Left: reducing increment spaces in parallel. Right: adjusting increment spaces when introducing a minimum level.  174
Figure 7.3  Results on Hermit for $d = 3$ without boundary using a constant minimum level.  199
Figure 7.4  Results on Hermit for $d = 3$ with boundary using a constant minimum level.  200
Figure 7.5  Results on Hermit for $d = 5$ and boundary points using 126 component grids.  201
Figure 7.6  Results on Hermit for $d = 5$ and boundary points using 426 component grids.  202
Figure 7.7  Results on Hermit for $d = 10$ and boundary points using 286 component grids.  203
Figure 7.8  Results on SuperMUC for $d = 3$ without boundary points using a constant minimum level.  204
Figure 7.9  Results on SuperMUC for $d = 5$ using boundary points and 456 component grids.  205
Figure 7.10  Results on SuperMUC for $d = 10$ using boundary points and 286 component grids.  206
Figure 7.11  Predictions for SuperMUC for $d = 5$ with boundary and constant minimum level.  207
Figure 7.12  Predictions for SuperMUC for $d = 10$ with boundary and constant minimum level.  207

**LIST OF TABLES**

| Table 3.1 | Leading term of the non-compulsory I/Os for the presented and previous results. | 33 |
| Table 3.2 | Non-compulsory I/Os of different algorithms for arbitrary dimensions. | 66 |
| Table 3.3 | Non-compulsory I/Os of different algorithms for two dimensions. | 75 |
| Table 3.4 | Non-compulsory I/Os of different algorithms for three dimensions. | 78 |
| Table 5.1 | Maximum refinement level and standard basic block size for the different dimensions. | 123 |
| Table 5.2 | Optimized versus unoptimized unidirectional hierarchization for 3 & 4 cores. | 125 |
Table 7.1 Complexities of the communication task of the sparse grid combination technique. 166
Table 7.2 Latency and bandwidth values used for the lower and upper bounds in the communication model. 187
Table 7.3 Number of component grids of the communication task of dimension d and level n. 191
Table 7.4 Rounds and makespan volume for different dimensions using no boundary points and a constant minimum level. 192
Table 7.5 Rounds and makespan volume for $d = 5$ using boundary points and a constant minimum level. 194
Table 7.6 Rounds and makespan volume for $d = 5$ using boundary points and an increasing minimum level. 194
Table 7.7 Total runtime split into latency and bandwidth term for Hermit using boundary points and a constant minimum level. 197
Table 7.8 Runtime predictions for different systems and $d = 5$ using boundary points and a constant minimum level. 198

LIST OF ALGORITHMS

Algorithm 3.1 One Jacobi iteration for the 2-dimensional 1-star stencil. 24
Algorithm 4.1 The unidirectional hierarchization algorithm for a component grid $C_\ell$. 111
Algorithm 4.2 The generalized unidirectional hierarchization algorithm for a set T. 113
Algorithm 5.1 Basic block optimized unidirectional hierarchization algorithm for a component grid $C_\ell$. 122
Algorithm 6.1 Divide and conquer hierarchization algorithm. 139
Algorithm 7.1 *Sparse Grid Reduce* for component grid $C_\ell$. 172
Algorithm 7.2 *SubSpace Reduce* for component grid $C_\ell$. 172
A universal phenomenon of modern computers is the memory bottleneck. A computer’s architecture is said to be limited by the memory bottleneck if it can process data much faster than it can access it. This bottleneck has developed as, over the last decades, the performance of the processors grew much faster than the latency and bandwidth with which the processors can access the data needed for their computations. The phenomenon is omnipresent in the sense that it holds in similar ways for single- and multi-core processors as well as for distributed systems. As a result, processors can rarely reach their peak performance as they are idling while waiting and starving for more data to process. When an algorithm cannot continue with its computations as the processors it runs on are waiting for data to be loaded or stored, it is said that the algorithm is memory-bound and limited by the memory bottleneck on these processors.

The standard way to lessen the memory bottleneck is the memory hierarchy, see Figure 1.1. Consider a single-core processor. At the beginning and at the end of the execution of an algorithm all data is stored in the large and slow main memory. To work with the data, it has to be loaded to the fast but small registers which can be accessed directly by the core. In between the main memory and the registers, several layers of caches are added. From main memory to registers, the caches are ordered by decreasing size and increasing speed. Although several layers of caches are added in practice, theory typically models only two layers of the memory hierarchy. In practice, the principal memory bottleneck is typically posed by the latency of and bandwidth between the smallest (and fastest) memory into which the problem fits completely and the next smaller and faster level of the memory hierarchy. The theoretical model focuses on these two layers which pose the principal memory bottleneck of the problem. In the theoretical model, the larger and slower memory is called main memory and the faster and smaller memory is called cache. These two terms can be used as synonyms for any two consecutive layers of the memory hierarchy. The following discussion also uses these two terms in that sense.

The additional, smaller and faster layers of memory allow algorithms to exploit two kinds of locality to increase the reuse of data and, by doing so, lessening the memory bottleneck: temporal locality and spatial locality. Temporal locality describes the reuse of data that is accessed repeatedly. If the data is still in the cache, it does not have to be loaded again from the main memory but can be accessed directly from the cache.
Spatial locality describes the use of data that is stored in nearby memory locations. As the memory is organized in blocks, so-called cache lines, all data items of a block are loaded into the cache if one of the items of the block is accessed. Hence, if a memory address of the same block is addressed in the near future, the data item already resides in the cache and does not need to be fetched from the main memory again. If the algorithms exploit locality, they can decrease the data transfer to main memory, and hence their dependence on the main memory bandwidth.

Multi-core processors typically share the large, slow layers of the memory hierarchy, i.e., the main memory and the L3 cache, while the fast and small layers are duplicated for each core. To exchange data between the different cores, the data has to be stored in the shared memory layers. As the bandwidth of the shared layers is identical regardless whether a single core or all cores of the multi-core processor are used for computations, the shared layers limit the performance of multi-core processors. Only when the data is kept in the higher, smaller and duplicated levels of the memory hierarchy, can speedups due to the increased total bandwidth be expected.

On distributed systems such as supercomputers, the different compute nodes exchange data by message passing. The bandwidth of the interconnection between the compute nodes is usually much lower than bandwidths of the memory subsystems on the nodes themselves. This adds, in addition to the memory hierarchy existing on each compute node, an additional layer of data transfer. Typically, this layer of communication via message passing is the slowest and hence further limits the performance of the algorithms.

There are, amongst others, three trends in computer design: 1) the number of cores of multi-core processors is increasing; 2) additional layers are added to the memory hierarchy; 3) larger and larger distributed systems are built. In consequence, designing communication efficient
algorithms that exploit spatial and temporal locality will gain more and more importance. In this thesis, communication efficient algorithms are meant to include both memory efficient algorithms optimizing the data transfer through the memory hierarchy as well as algorithms that minimize the communication that happens via message passing on distributed systems.

Increasing the peak performance of the processors, even to infinity, would not help to speed up memory-bound algorithms significantly as these algorithms are limited by the performance of the memory system. Instead, a faster access to the memory is beneficial as are algorithms that further exploit spatial and temporal locality. Hence, the classical complexity analysis of algorithms that counts the number of floating point operations is not suitable to predict the runtime of memory-bound algorithms. Instead, the number of cache misses or the total amount of data transferred between the main memory and the cache are a more accurate indicator for the runtime of those algorithms.

To precisely identify the algorithms for which the performance of the memory subsystems plays a crucial role, let us formally define memory-bound algorithms using ideas from the roofline model [WWP09]. For the definition, consider the operational intensity $I$ of an algorithm, i.e., the number of floating point operations the algorithm performs for each byte it transfers (typically between the main memory and the cache). Let $\pi$ denote the peak performance (in [flops]/[cycle]) of the processor and let $\beta$ denote its main memory bandwidth (in [bytes]/[cycle]). If $\pi > I \cdot \beta$, then the algorithm is said to be memory-bound on this processor, otherwise it is called compute-bound. For a memory-bound algorithm, the operational intensity is so low that the memory system cannot feed the processor in such a way that the processor can continuously work with peak performance. While there are many problems that allow for a high operational intensity, such as the multiplication of two dense matrices, there are even more for which the operational intensity is bounded by a small constant. Many problems from scientific computing, in particular solving partial differential equations (PDEs), can be formulated as sparse matrix-vector multiplications. The operational intensity of sparse matrix-vector multiplication is less or equal to that of dense matrix-vector multiplication and the latter is already bounded by a constant: for a dense matrix of size $n \times n$ the number of floating point operations is $2 \cdot n^2$ while $n^2 + n$ values have to be loaded to read the input. Hence, the operational intensity is less than $1/4$ for double precision and $1/2$ for single precision. This makes algorithms for sparse matrix-vector multiplication, and in particular for solving PDEs, prone to be memory-bound and limited by the memory bottleneck. To minimize the runtime of algorithms
solving these problems, it is crucial that the algorithms maximize the operational intensity by exploiting temporal and spatial locality.

The limitations imposed by the memory bottleneck become even more important as we aim to solve the problem at hand with constantly better accuracy. For PDEs, this means a finer step size of the discretization, which results in a larger sparse matrix and a larger coefficient vector. In general, the problems grow in size, which makes it harder to keep the cores busy through the memory bottleneck. For large problems, it is harder to exploit spatial and temporal locality to an extent that the given bandwidth suffices to feed the cores.

High-dimensional applications further reinforce this effect. As the number of grid points of a discretization depends exponentially on the dimension, an effect known as the “curse of dimensionality” [Bel61], the problems rapidly become too large to be solved efficiently with current methods. For PDEs, “high-dimensional” already means beyond the classical four dimensions, space and time. We hence need to rethink the discretization schemes as well as the algorithms working on these large data sets.

In scientific computing, a lot of effort is put into optimizing algorithms. This has resulted in many efficient libraries for standard tasks such as sparse matrix-vector multiplication as well as highly optimized code for specific applications. The runtime of algorithms from scientific computing, however, is rarely analyzed theoretically. Lower bounds for the complexity of the underlying problems are even rarer.

This work studies the complexity of different problems from scientific computing. The problems are analyzed in theoretical models that focus on the memory access and communication necessary to solve the problems, either through the memory hierarchy or by message passing. The problems studied come from the field of PDE solvers, one of the most important classes of numerical algorithms. In the classical setting of full grids, this work studies stencil computations, i.e., updating the grid according to a stencil. For high-dimensional settings, problems from the sparse grid combination technique are examined.

Analyzing the problems in the theoretical models instead of optimizing a given algorithm allows one to isolate the most important aspects and to come up with new algorithmic ideas. As the focus of this work are the communication aspects of the algorithms, this analysis identifies the limitations of current algorithms with respect to the memory bottleneck. The analysis also exhibits ways to circumvent the memory bottleneck. For the stencil computations, this work proposes blocking techniques and non-standard data layouts to reduce the number of cache misses. Through increasing spatial locality, the unidirectional hierarchization algorithm of sparse grids is implemented for component grids such that it performs within a factor of 1.5 of the lower bound given by the main memory bandwidth and the unidirectional principle. For
isotropic component grids, this thesis derives a divide and conquer approach for hierarchization that avoids the unidirectional principle, the dominating design pattern for sparse grid algorithms, on a global scale. The algorithm is cache-oblivious and avoids the global passes of the unidirectional principle which make any unidirectional algorithm inherently memory inefficient. Furthermore, the first systematic study of the communication step of the sparse grid combination technique is presented. The algorithms and their analysis are complemented with lower bounds for the respective problems that show that the presented algorithms are either optimal or within a small constant factor of the optimum.

Some of the new algorithmic ideas immediately led to efficient implementations for the given problem. Implementations are presented for the unidirectional hierarchization algorithm for component grids and the communication schemes for the sparse grid combination technique. For the stencil computations on full grids and the divide and conquer hierarchization algorithm for component grids, the theoretical analysis identified the limitations of the current algorithms and guided the development of possible algorithmic solutions.

All presented algorithms optimize the communication behavior of existing techniques, i.e., no new numerical methods are derived in this work. The altered, memory efficient algorithms perform the same floating point operations as their original counterparts and hence compute exactly the same results. For all computational problems, i.e., the stencil computations as well as both hierarchization algorithms, they also do not change the arithmetic circuit that describes the computation. In particular, the order of convergence of the numerical methods is not altered and thus not considered in this work. Instead, the computations are reordered and reorganized to create basic blocks that increase spatial as well as temporal locality. Moreover, the concepts of recomputation and divide and conquer are employed to reduce the data transfer.

All considered computational problems, i.e., the stencil computations as well as both hierarchization algorithms, can be phrased as sparse matrix-vector multiplications. Each of these problems gives rise to a certain type of sparse matrix with its own special structure. For the stencil computations and the implementation of the unidirectional hierarchization algorithm, the presented optimizations can be thought of as reorderings of these sparse matrices, i.e., permutations of the rows and columns, that create dense blocks and hence increase locality and therefore decrease data transfer. The lower bounds show that these permutations are the best possible given the structure of the respective matrices.

In the following sections, the results for the different problems are outlined in more detail. To do so, let $d$ denote the dimension of a given
grid, M the size of the cache and B the cache line size. Furthermore, it is always assumed that the input, for example an input grid $C_\ell$ of refinement level vector $\ell$, is significantly larger than the cache, i.e., $M \in o(|C_\ell|)$.

1.1 stencils on full grids

Stencil computations on low-dimensional grids are basic building blocks for many scientific applications including finite difference methods used to solve PDEs. The stencil describes how the value of a grid point is updated using the values of neighboring grid points, see Figure 1.2 for two examples. One step of a stencil computation updates all grid points once according to the stencil and thus performs few floating point operations per grid point. Accordingly, the operational intensity of stencil computations is typically low and these computations are therefore usually memory-bound.

Chapter 3 examines one step of a stencil computation for star stencils, such as the 5-point (2D) and 7-point stencil (3D), in the external memory model and the parallel external memory model. The (parallel) external memory models counts the number of I/Os; i.e., the number of read as well as write operations between the cache and the main memory. For stencil computations, the majority of the I/Os, and also the majority of the cache misses, occur to read the input and write the output. The I/Os to read the input and write the output are denoted as compulsory I/Os and all other I/Os are denoted as non-compulsory I/Os. This work analyzes the constant of the leading term of the non-compulsory I/Os for star stencil computations.

While optimizing stencil computations is an active field of research, there has been a significant gap so far between the lower bounds and the I/O complexity of the algorithms. For dimension $d \geq 4$, this work

--

1 For more details on the model of computation for a specific problem, refer to Chapter 2 and see also the chapter respective to the problem.
improves the lower bound between a factor of 4 and 6. For the upper bounds and \( d \geq 3 \), the constant of the leading term of the non-compulsory I/Os is analyzed for the first time, resulting in bounds that match up to a factor of \( d^{-\sqrt{d!}} \). For high dimension \( d \), this can be approximated as \( d^{-\sqrt{d!}} \approx d/e \). Furthermore, in three dimensions, the bounds match up to a factor of \( \sqrt{2} \) and improve the known results by a factor of \( 2\sqrt{3\sqrt{3}} \). In two dimensions, this work provides matching constants for the number of non-compulsory I/Os, thus closing a multiplicative gap of 4. All mentioned gaps have existed since 2002. The lower bounds split an arbitrary algorithm into rounds and bound the work per round with an isoperimetric inequality. The upper bounds improve upon the previous work by defining and using data layouts that exactly match the access pattern of the corresponding algorithm.

This work performs a theoretical analysis of stencil computations focusing on the I/O behavior in the scenario of the (parallel) external memory model. Before turning the theoretical insights into high-performance code for modern computer architectures, consider the following three aspects: first, an implementation on a modern computer system would need to take more effects into account than are modeled in the (parallel) external memory model. For example, an efficient implementation would need to be optimized for several layers of the memory hierarchy and may need to work with caches of limited associativity. Second, the theoretical improvements affect the non-compulsory term of the I/Os which is dominated by the compulsory term by a factor of \( \Theta(d^{-\sqrt{d}}) \). This limits the expected speedups to small percentages. Third, this work considers one single update of the whole grid according to the stencil and disregards the option to merge multiple stencil passes, i.e., multiple steps of the stencil computation. It is, however, likely that optimizations similar to the presented ones can also be applied to multiple layers of the memory hierarchy and multiple stencil passes.

1.2 Sparse grids and the combination technique

Solving high-dimensional numerical problems, in particular high-dimensional PDEs, is one of the great challenges of current and future high-performance computing (HPC) systems. Unfortunately, straightforward discretization schemes, such as full grids, fully suffer from the “curse of dimensionality”. This makes high-dimensional applications one of the compute-hungry drivers of exascale computing [DB+11].

Fortunately, hierarchical discretization schemes are of help: given certain smoothness conditions, a discretization based on sparse grids [Zen91] can reduce the number of grid points significantly while the accuracy of the solution only deteriorates slightly [BG04]. Sparse grids reduce the number of unknowns by choosing a suitable basis, the hierarchical basis, and then selecting the most important basis function of the hierarchical
introduction

Figure 1.3: The complicated data dependencies in a sparse grid (left) due to the hierarchical basis. The arrows depict the hierarchical predecessor relation in x-direction (black) and y-direction (red). Two hierarchical basis functions overlap, i.e., depend on each other, if there is a directed path between them. Thus, many more basis functions overlap although they are not directly connected. In contrast: in the nodal basis of a full grid (right, in small), only directly neighboring (also diagonally) basis functions overlap.

basis. While sparse grids lessen the curse of dimensionality, they are not able to break or avoid the curse completely.

The base change from the full grid basis to the hierarchical basis is crucial for sparse grids. It is called hierarchization and is one of the most fundamental algorithms for sparse grids. While the hierarchical basis enables a reduction of the curse of dimensionality, it also introduces more complicated data dependencies to non-neighboring grid points as depicted in Figure 1.3. As a result, only very few algorithms have been implemented to work directly in the hierarchical basis of sparse grids. Those that have take advantage of the unidirectional principle. The unidirectional principle breaks the complicated d-dimensional dependencies of the hierarchical basis by splitting the problem into d phases. Then, in each of the phases, the unidirectional principle considers only 1-dimensional subproblems.

Instead of working directly in the hierarchical basis of sparse grids, there is another common approach to sparse grids: the sparse grid combination technique [GSZ92] depicted in Figure 1.4. The sparse grid combination technique, or simply combination technique, is an extrapolation scheme. It solves the original problem formulation for many, but coarse and anisotropic, i.e., refined differently in different dimensions, regular
1.2 SPARSE GRIDS AND THE COMBINATION TECHNIQUE

Grids also called component grids [Heg03a]. A suitable linear combination of the component grid solutions then retrieves a single solution in the hierarchical sparse grid space. These smaller subproblems on the component grids can be solved independently and therefore in parallel. The hierarchical approach thus overcomes a central problem of massively parallel computations: the splitting into the component grids ensures scalability on future high-performance computers by breaking the global communication requirements of conventional discretization approaches. Furthermore, employing the combination technique there is no need to change the application code if it can be applied to arbitrary anisotropic and regular grids.

This work focuses on the combination technique and sparse grid algorithms applied to the component grids of the combination technique. It does not derive algorithms specifically designed for regular or adaptive sparse grids but discusses, whenever applicable, how the presented approaches also generalize to these grids.

1.2.1 Unidirectional Hierarchization

Hierarchization is one of the most fundamental tasks for sparse grids and additionally an essential preprocessing step for one of the communication schemes presented in Chapter 7. Hierarchization describes the transformation from the full grid basis to the hierarchical basis of sparse grids. As for most sparse grid algorithms, the implementations of the hierarchization algorithm up to now take advantage of the unidirectional principle splitting the algorithm into \(d\) distinct phases. Each single phase then deals only with 1-dimensional subproblems.

Chapter 5 discusses a memory-efficient implementation of the unidirectional hierarchization algorithm for the component grids of the combination technique. The algorithm exploits the additional structure of the component grids, compared to regular or adaptive sparse grids, to simplify the navigation on the grid. In addition, it reorders the computations to create large basic blocks, which can also be parallelized easily. The derived implementation runs within a factor of 1.5 of the runtime.
achievable for large grids by any hierarchization algorithm implemen-
ting the unidirectional principle. The implementation outperforms the
currently fastest generic software StructuredSG [BJPM12] by a factor be-
tween 5.8 and 41 for problems larger than 30MiB, and up to the maximal
tested problem size of 8GiB.

To achieve this performance gain, the algorithm is optimized for the
anisotropic component grids and cannot be applied to regular or adap-
tive sparse grids. Although theoretic considerations guided the opti-
mizations, Chapter 5 does not derive the exact I/O complexity of the
algorithm but focuses on an efficient implementation.

1.2.2 Divide and Conquer Hierarchization

After a nearly memory-optimal unidirectional hierarchization algorithm
has been presented in Chapter 5, the question whether we can beat the
unidirectional lower bound arises naturally. Can we reduce the number
of cache misses by avoiding the \( d \) global sweeps of the unidirec-
tional principle?

Chapter 6 derives a divide and conquer hierarchization algorithm for
isotropic component grids, i.e., component grids that are refined equally
in all dimensions. The algorithm is cache-oblivious and avoids the uni-
directional principle globally. Instead, it applies it recursively to smaller
subproblems using the divide and conquer strategy. Assuming a tall
cache of size \( M = \omega(B^d) \), this approach reduces the leading term of
the number of cache misses by a factor of \( d \) compared to the unidirec-
tional lower bound.\(^2\) Hence, this algorithm is optimal with respect to
the leading term of the cache misses as the constant of the leading term
is now down to 1 and therefore trivially matched by a scanning lower
bound. Chapter 6 further complements the algorithm with a non-trivial
lower bound for the second term of the number of I/Os. This lower
bound is restricted to algorithms that can be expressed as linear arith-
metic circuits, i.e., algorithms that compute linear combinations of the
input. Algorithms that compute arbitrary algebraic or transcendental
functions of the input are disallowed. For \( B = 1 \), a minor modification
of the analysis of the algorithm shows that the lower and upper bound
also match asymptotically with respect to that second term.

The presented upper bound introduces a new algorithmic idea for
sparse grids, i.e., it uses the divide and conquer technique to avoid the
unidirectional principle globally. When implementing this algorithm,
one has to be careful, though. In the current version, the coarse levels
of the recursion are treated in a brute force manner in the algorithmic
design as well as in the analysis. Hence, the complexity of the algorithm
can be worse than that of the unidirectional algorithm when the tall

\(^2\) As common in the sparse grid literature, the dimension \( d \) is assumed to be constant in the
\( O \)-notation.
1.2 Sparse Grids and the Combination Technique

Sparse grids and the combination technique allow us to solve each time step of the original problem independently and hence in parallel on the component grids by breaking the global communication requirements of conventional discretization approaches.

Between the time steps, or at least every few time steps, it is, however, necessary to combine the component grid solutions (reduce), and then to distribute the joint solution back again (broadcast). This introduces a synchronization barrier and requires a reduced but global communication as illustrated in Figure 1.5.

Chapter 7 derives two optimal communication schemes for this remaining communication/synchronization bottleneck. While the first scheme is designed to minimize the number of communication rounds, the second scheme is optimal with respect to the total communication volume. Using the derived strategies, measurements on HPC systems for 3 dimensions, 5 dimensions, and an extended 10-dimensional setting, are conducted. Furthermore, a communication model that is well-suited to predict the cost of the communication step is presented. Given performance characteristics of the employed HPC system, this model estimates lower and upper bounds for the runtime of the communication step.
model can also be applied to settings that are as yet out of scope due to the high computational demand, for example to problems in 10 dimensions, and to predict their communication feasibility for future HPC platforms.

The only communication operations carried out by the communication schemes are *AllReduce* operations, and hence the optimality of the communication schemes depends on the optimality of the implementation of *AllReduce*. For the analysis of the communication schemes, it is assumed that *AllReduce* works in two distinct phases, reduce and broadcast. While this assumption facilitates the analysis of the communication schemes and the derivation of the lower bounds, it is not crucial for the optimality of the communication schemes. If a communication scheme is optimal with respect to a certain cost measure for separate reduce and broadcast, and *AllReduce* is implemented such that it minimizes that same cost measure without the assumption of separate reduce and broadcast, then the communication scheme is also optimal with respect to that cost measure without the assumption of separate reduce and broadcast. In short: as *AllReduce* is the only communication operation the communication schemes perform, they inherit optimality with respect to a certain cost measure from the *AllReduce*-operation.

1.3 Outline of the thesis and research contributions

This thesis is organized as follows: Chapter 2 gives an overview of different communication models from the literature. Chapter 3 analyzes star stencil computations. Then, background and notation regarding sparse grids and the combination technique is introduced in Chapter 4. Subsequently, an efficient implementation of the unidirectional hierarchization is discussed in Chapter 5, and Chapter 6 derives and analyzes the divide and conquer hierarchization algorithm. Chapter 7 addresses the communication schemes for the combination technique and Chapter 8 concludes this work.

While some of the results have already been published [HJ12, HJ13, Hup14b, Hup13, HJH+14], some are parts of ongoing projects [HJ14b, HJ14a, HHJP14]. Most of the research was done in collaboration with different co-authors, namely Riko Jacob, Mario Heene, Markus Hegland, and Dirk Pflüger. This thesis contains large text parts of these projects, either of the published paper or of the current draft of the unpublished manuscript. I will discuss the relation of this thesis to other published and unpublished manuscripts in further detail at the end of the introduction of every chapter. In particular, Chapter 7 and parts of the background on sparse grids discussed in Chapter 4 are joint work with Mario Heene. It is planned that these contents will also be covered in his PhD thesis. In addition, as the communication model from Chapter 7 builds
upon the communication models from the literature discussed in *Chapter 2*, parts of *Chapter 2* might also appear in his thesis.
All communication models discussed in this chapter are taken from the literature and are well established. The models split into those that focus on the memory transfer through the memory hierarchy (Section 2.1) and models that rely on message passing and hence can be applied to distributed systems (Section 2.2). Each model is designed to capture and isolate the most important aspects for communication efficient algorithms on real world systems. None of the models captures all or even most of the factors involved. If the model would do so, the model would grow so complex that it would be very complicated to analyze the complexity of an algorithm. Instead, the models are designed to be simple with the purpose to allow a theoretical analysis of the problem at hand, focusing on the aspects that are most likely to dominate. We use the standard literature models for our analysis and, when reasonable for the problem at hand, state minor modifications to the model in the according chapter.

The main models we use are the I/O model (Section 2.1.1), the external memory model (Section 2.1.2), the parallel external memory model (Section 2.1.3), the cache-oblivious model (Section 2.1.4) and the bulk-synchronous parallel model (Section 2.2.1). The stencil computations (Chapter 3) are analyzed in the (parallel) external memory model which is an extension of the I/O model. The divide and conquer hierarchization algorithm (Chapter 6) is analyzed in the cache-oblivious model which assumes that the parameters of the external memory model are not known to the algorithm. The communication schemes for the reduce/broadcast step of the sparse grid combination technique (Chapter 7) need to be analyzed in a distributed setting and hence we employ a variant of the bulk-synchronous parallel model. To make sense, all models have to assume that data items are atomic and cannot be compressed.

Research contributions. This chapter discusses standard communication models from the literature and its contents have been presented with different foci in the following publications and ongoing projects, upon which this work builds: [HJ13, HJ12, HJH+14, HJ14b, HJ14a, HHJP14]. These projects were and are joint work with Riko Jacob, Markus Hegland, Mario Heene and Dirk Pflüger. This chapter contains text parts from the different publications and manuscripts that were among my contributions to the according publication or manuscript. As the communication models provide the background necessary to an-
alyze the communication schemes of Chapter 7, parts of this chapter mights also appear in the PhD thesis of Mario Heene.

2.1 MODELS FOR THE MEMORY HIERARCHY

The models of this section describe the memory hierarchy, typically two levels of it. The first, smaller and faster level is either called internal memory or cache. The second, larger and slower level is either called external memory or main memory. Between these two levels of the memory hierarchy the data is transferred in blocks, also called cache lines. We use both triples of terms synonymously, i.e. we may talk about loading a cache line from external memory. We first discuss the memory models and then discuss their limitations in Section 2.1.5.

2.1.1 I/O Model

One of the first models to measure the I/O-complexity of algorithms was the red-blue pebble game, also called I/O model, of Hong and Kung [HK81]. Hong and Kung assume that there are two levels of memory, an external memory of infinite size on which all data is stored initially, and an internal memory of size \( M \) to which the data has to be loaded to perform computations. Single data items, i.e., blocks of size \( B = 1 \), can be moved between the internal and the external memory by I/O operations. An I/O operation either transfers a single data item from external to internal memory (read) or from internal to external memory (write). At the beginning and at the end of the algorithm, all data has to be stored in external memory. The I/O-complexity in the red-blue pebble game is the number of data items that need to be transferred between internal and external memory. As for all models simulating the memory hierarchy, the computations that are done once the data resides in internal memory are not considered. As the model is limited to \( B = 1 \), it focuses on temporal locality only.

The data transfer between external memory and internal memory is managed explicitly by the algorithm. This has three consequences: first, the cache-replacement strategy can be determined by the algorithm. Second, blocks can be deleted within internal memory and do not have to be stored back to external memory to free space in internal memory. Third, data can be written directly to blocks in external memory without loading these blocks to internal memory first.

2.1.2 External Memory Model

Aggarwal and Vitter generalized the red-blue pebble game to the external memory model (EM model) [AV88] depicted in Figure 2.1. The EM model accounts for spatial locality by dividing the external memory into blocks
of size $B \geq 1$. In this model, an I/O operation always affects a whole block by either loading a block of $B$ items from external memory to internal memory or by storing $B$ data items from internal memory to a block of the external memory. As for the I/O model, the data transfer between external and internal memory is managed explicitly.

2.1.3 Parallel External Memory Model

Arge et al. generalized the EM model to the parallel external memory model (PEM model) depicted in Figure 2.1. In the PEM model there are $P$ processors that each have their own private cache of size $M$. In addition, the $P$ processors share an infinitely large external memory. The processors cannot communicate directly but have to exchange data by writing it to and reading it from the external memory. One parallel I/O can transfer (read or write) one block of data of size $B$ for each processor simultaneously. As before, the data transfer is again managed explicitly. Concurrent writes that store different elements of the same block of external memory are, however, not possible.

2.1.4 Cache-Oblivious Model

The I/O model, the EM model and the PEM model have in common that they are cache-aware: the cache size $M$ as well as the block size $B$ are known to the algorithm and can be exploited to select subproblems
that are small enough to be handled efficiently with the given internal memory. In contrast, the parameters $M$ and $B$ are not known to the algorithm in the cache-oblivious model [Pro99, FLPR99]. The idea is to design algorithms that work efficiently for any $M$ and $B$ and hence efficiently across different machines. As the internal memory size and the block size are unknown to an algorithm, the cache has to be managed implicitly. To do so, the cache-oblivious model assumes an “ideal-cache” and in particular that the best (offline) cache-replacement strategy is used.

In this work, we also focus on the most common scenario of a write-back cache with write allocation. A write-back cache updates the values of data items that are already in the cache just in the cache and stores the altered values of a block only to external memory when the block is evicted from cache. A write allocation policy always loads a block from external to internal memory when one of the items of the block are updated. In this case, it is not possible to directly update values in the external memory without loading the according block to the internal memory first. Therefore, a cache miss occurs if an instruction refers to a data item currently not in cache. If that happens, this data item needs to read from external memory causing a read I/O. While an I/O can either be a read or a write operation, cache misses only refer to read operations. Due to the write allocation policy, however, a write operation can trigger a read and therefore a cache miss, if the data item to be updated has not been in cache beforehand. When we refer to cache misses due to write operations, we refer to this phenomenon and not to the write operation itself.

This work uses the term cache misses also without the context of the cache-oblivious model and often synonymously for I/Os, as cache misses and I/Os are very similar. As I/Os and cache misses differ, however, the appropriate term is always used when it is important, i. e., when the constant of a stated complexity matters. When important, this work assumes an implicitly managed write-back cache with write allocation when it talks about cache misses and an explicitly managed cache when it talks about I/Os.

2.1.5 Limitations of the Models for the Memory Hierarchy

There are several limitations that are shared by all models that describe the memory hierarchy. These limitations include: first, all models disregard the computations that are performed once the data is stored in internal memory and only account for data that is transferred between the different levels of the memory hierarchy. If a problem is compute intensive disregarding all local computations is likely to oversimplify the analysis. This assumption is well justified, however, if the problem involves few computations and the algorithm is memory bound. Sec-
2.1 Models for the Memory Hierarchy

Second, the cache replacement strategy can either be explicitly managed by
the algorithm (I/O, EM and PEM model) or is assumed to be the best
possible (cache-oblivious model). If we can explicitly manage the cache-
replacement strategy, we can in particular use the best possible strategy.
In practice, using the best possible (offline) strategy is usually not pos-
sible. The best cache replacement strategy, however, can be simulated
with a least recently used (LRU) strategy while usually only increasing
the cache misses by a small constant factor [FLPR99]. As we analyze
the constant of the leading term of the non-compulsory I/Os for sten-
cil computations in Chapter 3 and the constant for the total number of
I/Os for the divide and conquer hierarchization algorithm in Chapter 6,
such a simulation would disturb the presented results. The presented
algorithms, however, essentially employ a LRU strategy such that the as-
sumption of using the best cache replacement strategy is not important
for them.

Besides the cache replacement strategy, an explicitly managed cache
has other consequences. The user can also choose how to evict data
from internal memory. If a value is not needed or wanted anymore,
it can be deleted directly within internal memory and does not need
to be stored back to external memory. Hence, I/O operations to free
the internal memory can be saved if the data is not needed at a later
point. Also, data can be written directly to blocks in external memory,
without loading these blocks to internal memory first. This approach
is similar to a write-through cache with no-write allocation. The key
about an explicitly managed cache is that we can mix a write-through no-
write allocation policy with a write-back write allocation policy. While
it is possible to mix both strategies in the models, we do not exploit
this in Section 2.1.2 when we analyze the stencil computations in the
(P)EM model. Instead, we discuss different versions of writing policies
separately in Section 3.8.

In addition, the cache is assumed to be fully associative. While com-
mon in practice, we do not address caches of limited associativity in this
work.

Also, the discussed memory models focus on two levels of the mem-
ory hierarchy. There are, however, various approaches to extend the ex-
ternal memory model to several layers of the memory hierarchy [ACS87,
ACFS94, SCD02].

The parallel PEM model is limited to the scenario in which multiple
cores share some sort of external memory to exchange data. Graphic
processing units (GPUs) and non-uniform memory access (NUMA) ar-
chitectures can be seen as versions of this scenario.
2.2 MODELS FOR DISTRIBUTED SYSTEMS

This section discusses communication models that rely on message passing and therefore apply to distributed systems. In particular, these models apply to the supercomputers we use in Chapter 7 to conduct the experiments regarding the reduce/broadcast step of the sparse grid combination technique. After discussing the different models, we discuss their limitations in Section 2.2.4.

2.2.1 Bulk-Synchronous Parallel Model

The bulk-synchronous parallel model (BSP model) of Valiant [Val90] assumes that processors communicate directly by exchanging messages. The BSP model is synchronized as it proceeds in supersteps dividing the algorithm into parts. In each superstep a processor can perform local computations as well as send and receive messages. Only if every processor has finished its task for the current superstep, all processors advance to the next superstep. Furthermore, communicating a message incurs two kinds of costs: a latency or startup term independent of the message size and a bandwidth term such that the communication of larger messages takes longer. The BSP model also takes the cost of local computations into account, such that the cost of a superstep is made up of these three components. The runtime of an algorithm is the sum of the runtimes of all its supersteps.

2.2.2 LogP Model

The logP model of Culler et al. [CKP+93] builds upon the BSP model. Processors communicate by point-to-point messages and in the simplest form the messages are single or a small number of words. The model is asynchronous and has four main parameters: \( P \) is the number of processors. The overhead \( o \) describes the time a processor takes to send or receive a message and during this time no other operation can be carried out. Once a message has been sent, the upper bound \( L \) on latency states how long a message takes at most to arrive at its destination. The destination processor then has to receive the message. Hence, sending and receiving a message takes at most \( o + L + o \) time. In addition, the model uses a gap parameter \( g \) as the minimum time between two message transmissions or receptions at a processor. The gap \( g \) can be seen as the inverse of the bandwidth. If \( o \geq g \), the gap can be eliminated from the model. The total communication cost of an algorithm in the logP model is defined to be the length of the critical path of the transmitted and received messages.
2.2.3 MapReduce Model

The MapReduce model by Dean and Ghemawat [DG08] is designed for easy parallelization of algorithms running on very large data sets. A computation in the MapReduce model proceeds in rounds, alternating local computations done in parallel with a parallel communication phase. The basic items in the MapReduce model are \(<key, value>\) pairs. Each round consists of three stages: a map, a shuffle and a reduce stage. In the map stage, a mapper takes one \(<key, value>\) pair as input and produces a set of intermediate \(<key, value>\) pairs as output. As the input to the map function is a single \(<key, value>\) pair, the map function is easily parallelized across different processors for several \(<key, value>\) pairs. In the shuffle stage the data is redistributed onto the processors such that each processor holds all values associated with the same intermediate key. In the reduce stage, the reducer takes all values associated with one intermediate key to produce a (possibly smaller) set of values (still associated with the same key). All the reduce operations of distinct keys can be executed in parallel. However, the map stage has to finish before the reduce stage can start as the latter needs access to all intermediate \(<key, value>\) pairs of the same key before it can start.

There are several ways to define the complexity of an algorithm in the MapReduce model [GSZ11]. The simplest complexity measure is the number of rounds. The MapReduce model, however, allows sequential, one round algorithms by mapping all inputs to a single key. To eliminate this possibility, the capacity of the reducers can be limited to \(\mathcal{O}(N^{1-\epsilon})\) for a problem of size \(N\).

2.2.4 Limitations of the Models for Distributed Systems

The models of this section rely on message passing and are hence applicable for distributed systems. To keep the models simple, they therefore disregard the memory hierarchy. The assumption is that the bandwidth of the communication channel used for message passing is much smaller than the main memory bandwidth and the cache bandwidths available on the processors themselves. Hence, the bandwidth available for the message passing operations constitutes the crucial bottleneck which allows to disregard all local operations, including the local data transfer through the memory hierarchy.

In addition, all discussed models for distributed systems disregard the topology of the communication network as well as the routing algorithm used to transmit the messages. They assume that all nodes can communicate with each other for the same cost. In practice, the bandwidth as well as the latency between two nodes typically depends on the pair of nodes that is communicating. Furthermore, some of the models assume that the communication is round based which is also
usually not the case in practice. Allowing asynchronous execution of an algorithm, however, can only speed up its execution. Both simplifying assumptions, synchronous rounds and a uniform interconnect between all nodes, are usually not crucial for the predictive power of the models. As an example, consider the communication model used to analyze the reduce/broadcast step of the sparse grid combination technique in Section 7.3. This model also assumes that all nodes are uniformly interconnected and that the communication is round based. This model accurately predicts the runtime of the communication schemes despite these simplifying assumptions as the experiments conducted in Section 7.3 are going to show.
3.1 INTRODUCTION

Stencil computations are the most performance critical component for many tasks in scientific computing. In particular, they appear when PDEs are solved. In order to solve a PDE with numerical methods, the space needs to be discretized and a standard discretization method for low dimensional Euclidean spaces are regular grids. The differential operator can then be turned into a linear function of a grid point and its neighbors by a finite difference method. Such a linear function is also called stencil and results in a very regular sparse system of linear equations. To make use of the sparsity, such systems are typically solved with iterative solvers like the Jacobi or Gauss-Seidel method. The kernel of these methods is the evaluation of the underlying stencil.

To clarify how stencils are used to solve PDEs we give a simple example. Consider the one-dimensional heat equation which describes the variation of temperature on a pole over time. For a function $u(t, x)$ describing the temperature of the pole at time $t$ and position $x$, this problem can formally be written as the PDE $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$. We approximate the pole by a one-dimensional grid where each grid point stores the temperature of the pole at the respective point. Using an explicit finite difference method, the temperature of the grid points at time $t + \Delta t$ can be computed given the temperature at time $t$. The PDE is approximated by

$$\frac{u(t+\Delta t, x) - u(t, x)}{\Delta t} = \frac{u(t, x-\Delta x) - 2u(t, x) + u(t, x+\Delta x)}{(\Delta x)^2}.$$  

Abbreviating $c := \frac{\Delta t}{(\Delta x)^2}$, this solves to

$$u(t+\Delta t, x) = c \cdot u(t, x-\Delta x) + (1 - 2c) \cdot u(t, x) + c \cdot u(t, x+\Delta x)$$

which in turn gives rise to the one-dimensional 1-star stencil.

Hereby the stencil states which neighboring vertices of a grid point are necessary to update the grid point. The task is to recompute the values at the vertices of the grid according to the stencil. Another well known
Algorithm 3.1 One Jacobi iteration for the 2-dimensional 1-star stencil on the input array $A[k_1][k_2]$ and the output array $B[k_1][k_2]$. Truncation of the stencil at the boundary is disregarded.

```
for $i \leftarrow 1$ to $k_1$ do
    for $j \leftarrow 1$ to $k_2$ do
        $B(i, j) = -4 \cdot A(i, j) + (A(i-1, j) + A(i+1, j) + A(i, j-1) + A(i, j+1))$
    end for
end for
```

Example is the linear approximation of the Laplacian on a regular two-dimensional grid of mesh width $h$ as given by

$$
\Delta u(x, y) \doteq \frac{1}{h^2} \left[ -4u(x, y) + u((x-h), y) + u((x+h), y) + u(x, (y-h)) + u(x, y+h) \right].
$$

This defines the so-called 5-point or 1-star stencil depicted in Figure 1.2. This stencil can then be used in a Jacobi iteration to compute one time step for the 2-dimensional heat equation as exemplified in Algorithm 3.1.

Stencil computations are typically memory bound as they perform relatively few floating point operations on the data. The theoretically available peak floating point performance cannot be achieved because the memory system is the bottleneck limiting the speed. Hence, optimizing the memory access has become the main focus when designing high-performance stencil-code. We employ the external memory model [AV88] and the parallel external memory model [AGNS08] to count the number of memory accesses.

For stencil computations, the classical asymptotic analysis is too coarse to give interesting insights as the majority of the I/O operations is already needed for reading the input and writing the output. In fact, many simple algorithms for the 5-point stencil are within a factor of 5 of this lower bound. I/O operations related to the initial read of the input and the final write of the output are called compulsory I/Os or cold cache misses. All other I/Os are called non-compulsory I/Os or capacity misses (because they are unnecessary for sufficiently large main memory $M$).

This work examines the constant of the leading term of the non-compulsory I/Os caused by one update of the grid according to the $s$-star stencil in the external memory model and the parallel external memory model. Any naïve algorithm performing a stencil computation achieves the correct asymptotics for the total number of I/Os. Analyzing the constant of the total number of I/Os enables us to determine whether the algorithms exploit locality generally. But to quantify how well locality is exploited exactly, we need to drill down to the leading term of non-compulsory I/Os. The asymptotics of the leading term of the non-
compulsory I/Os determines whether an algorithm exploits the data layout it works on (as the presented memory efficient band algorithms) or is not able to do so (as standard blocked algorithms working on a row- or column-major data layout). Examining the constant of the leading term of the non-compulsory I/Os enables us to determine the efficiency of different data layouts and the corresponding algorithms.

In two dimensions, matching lower and upper bounds are given closing a multiplicative gap of 4. In three dimensions the provided bounds match up to a factor of $\sqrt{2}$ improving the known results by a factor of $2\sqrt{3}\sqrt{B}$. For dimensions $d \geq 3$, the lower bound is improved between a factor of 4 and 6. For arbitrary dimension $d$, the first analysis of the constant of the leading term of the non-compulsory I/Os is presented. For $d \geq 4$ the lower and upper bound match up to a factor of $\frac{d-1}{\sqrt{d!}}$. For high dimensions $d$, this can be approximated as $\frac{d-1}{\sqrt{d!}} \approx \frac{d}{e}$.

The lower bound combines a round argument with an isoperimetric inequality to bound the progress that can be achieved in each round. The isoperimetric results needs to be deduced and adapted carefully, using a pathwidth argument amongst others, such that no constant factors are lost in the analysis.

For the complexity of the upper bounds it is crucial that the algorithms work on data layouts that exactly match their access patterns. This work formalizes such layouts as band layouts and develops a framework for the corresponding (memory efficient) band algorithms. In summary, the algorithms work as follows: a $d-1$ dimensional sweep shape is swept through the grid creating work bands which are evaluated one after another. To be able to evaluate the whole grid, these work bands need to overlap. This overlap of the work bands gives rise to the data layout. Vertices that belong to distinct sets of overlapping work bands need to be stored in distinct blocks. Vertices that belong to the same set of work bands are aggregated to so called $k$-intersections and stored in contiguous memory. Evaluating a work band can then be done by sweeping through several $k$-intersections simultaneously. The vertices that are accessed by more than one work band determine the number of non-compulsory I/Os.

Stencil computations can be thought of as a sparse-matrix vector multiplication where the large, sparse matrix is defined by the stencil. The structure of the sparse matrix determined by the stencil can be exploited to find memory efficient algorithms performing this multiplication. To do so, it is crucial to find permutations of the matrix that create dense blocks as dense blocks can be applied efficiently. The algorithms and data layouts presented in this chapter provide such reorderings.

Before turning the theoretical insights into high-performance code for modern computer architectures, consider the following three aspects: first, the compulsory misses dominate the non-compulsory misses by a factor of $\Theta\left(\frac{d-1}{\sqrt{M}}\right)$, i.e., for every non-compulsory miss there are
\(\Theta\left( d - \sqrt{M} \right)\) compulsory cache misses. Hence, the expected speedups from the presented optimizations are limited to small percentages unless the cache is very small, e.g. a register.

Second, this work examines one single update of the grid according to the stencil. Optimizing multiple stencil passes at once is common in practice and can be modeled by introducing a temporal dimension. When a temporal dimension is included, the compulsory I/Os no longer dominate the non-compulsory ones as the number of computations is the product of the spatial and temporal dimensions whereas the number of compulsory I/Os solely depends on the spatial dimensions. Hence, for multiple stencil passes, increasing temporal (and spatial) locality can speedup the code significantly. While this chapter does not explicitly address a time step setup, the implications of the presented work for a setup including a time step are discussed in the conclusions (Section 3.9).

Third, this work focuses on the theoretical I/O behavior of stencil computations. The analysis is limited to two levels of the memory hierarchy and assumes, as usual for the external memory model, a fully associative cache. When implementing stencil computations, one may need to be careful about theoretic I/O behavior, which this work studies, and optimizations that improve runtime on current computer architectures. Some of the data layouts presented in this work, in particular the 3-dimensional hexagonal band layout, are complex and may not be suitable for implementation as they could require sophisticated padding schemes or might interfere with prefetching. In general, they could increase runtime although the number of cache misses is reduced. Implementing the presented algorithm is out of the scope of this work as this work improves the theoretic I/O behavior of stencil computations. However, diagonal hyperspace cuts, similar to the ones proven optimal in this work, are often employed in empirical work to select suitable substructures for computation.

The chapter proceeds by defining the theoretical model and problem precisely, presents the results and discusses related work. In Section 3.5 the lower bounds are derived. The lower bound section first proofs an isoperimetric result and analyzes the isoperimetric sets. After stating the relevant concepts of pathwidth these findings are combined to the lower bound. The algorithmic framework for the upper bounds as well as the notation required for the upper bounds is given in Section 3.6. Then, the upper bounds which are inspired by the lower bounds are described in Section 3.7. Section 3.8 discusses alternative theoretical models and their implications upon the presented bounds. Section 3.9 concludes this chapter.

**Research Contributions.** The results presented in this chapter are joint work with Riko Jacob. The lower bounds have been published in conference proceedings [HJ13]. A draft including the upper
problem definition

3.2.1 Computational Model

The computational model we consider in this chapter is the external memory (EM) model of Aggarwall and Vitter [AV88] with internal memory size \( M \) and block size \( B \) as detailed in Chapter 2. To analyze the stencil computations, we classify the I/Os into compulsory I/Os (cold misses), which account for the first access to a block and writing the final output, and non-compulsory I/Os (capacity misses). Non-compulsory I/Os are due to the limited size of the internal memory. In the EM model the cache is always assumed to be fully associative and hence conflict misses do not occur.

In the EM model, the internal memory is managed explicitly. This has three major implications. First, the cache replacement strategy can be specified by the user and hence assumed to be optimal. Second, it is up to the user how data is evicted from internal memory. The data can either be stored back to external memory or deleted within internal memory without causing an I/O operation. Third, data can be written directly to blocks in external memory, without loading these blocks to internal memory first.

We further assume that all I/Os are simple, i.e. data elements are moved instead of copied between internal and external memory. While this facilitates the derivation of our bounds, this assumption is not crucial and matching bounds assuming simple I/Os translate to matching bounds using non-simple I/Os as we discuss in Section 3.8.

3.2.2 Task

The task we consider is to perform one update of all values of a grid according to the \( s \)-star stencil. For the basic notation let \([k]\) abbreviate \(\{0, \ldots, k-1\}\), let \([k_1] \times \ldots \times [k_d]\) denote the \(d\)-dimensional grid and \(\mathbb{Z}_{k_1} \times \ldots \times \mathbb{Z}_{k_d}\) the \(d\)-dimensional torus of side lengths \(k_i\). Denote by \(\|\cdot\|_1\) the \(\ell^1\)-norm which is defined as usual for the grid. For an element \(v \in \mathbb{Z}_{k_1} \times \ldots \times \mathbb{Z}_{k_d}\) of the torus it is given by

\[
\|v\|_1 = \sum_{i=1}^{d} \min\{(-v_i \mod k_i), (v_i \mod k_i)\}.
\]

Denote by \(V\) the vertices of the grid or torus.
We consider out-of-place computations. Hence, there is an input layer \( V_{in} := V \times \{in\} \) and an output layer \( V_{out} := V \times \{out\} \) of the grid or torus. Initially, each vertex of the input layers stores a value while the output layer is empty. At the end of the computation, the values updated according to the stencil have to be stored in the output layer \( V_{out} \). The function which maps the values of \( V_{in} \) to \( V_{out} \) is described by a stencil. The task is to evaluate the stencil for all points of the output layer, i.e. to compute all values of the output grid and to write these results to external memory.

We consider so called \( s \)-star stencils. Denote by \( v_{in} \in V_{in} \) and \( v_{out} \in V_{out} \) corresponding vertices of the input and output layer, i.e. the first \( d \) coordinates of these vertices of the \( d \)-dimensional grid or torus are identical. The \( s \)-star stencil \( S_s \) for a vertex \( v_{out} \in V_{out} \) is defined as all vertices within distance \( s \) from \( v_{in} \),

\[
S_s(v_{out}) := \{ w \in V_{in} : ||w - v_{in}||_1 \leq s \}.
\]

This also implies that the stencils are cut off at the boundary of the grid as shown in Figure 1.2. For the asymptotic notation we assume throughout the chapter that \( s \) is a small constant.

The computation graph \((V_{in} \cup V_{out}, E)\) for the \( s \)-star stencil is obtained by, for all vertices in \( v_{out} \in V_{out} \), connecting the input layer to the output layer via adding the edges \((w, v_{out})\) for all vertices \( w \in S_s(v_{out})\):

\[
E := \{(w, v_{out}) \in V_{in} \times V_{out} : ||w - v_{in}||_1 \leq s \}.
\]

The 1-star stencils are the most common stencils. Since upper (lower) complexity bounds for the \( s \)-star stencil induce upper (lower) bounds for all stencils which are subsets (supersets) of the \( s \)-star stencil meaningful choices also include \( s = 2 \) and \( s = 3 \).

Working out-of-place on an input and output layer of the grid is not essential for neither the lower nor the upper bounds but simplifies their analysis. When we later argue about the stencil computations, the distinction between input and output layer is less strict. We say to evaluate a vertex \( v \) of the grid (torus) \( V \) when we compute the stencil for \( v_{out} \) and have the input \( S_s(v_{out}) \subset V_{in} \) in internal memory. Section 3.8 discusses the implications when we want to work in-place.

We consider computing the value for one grid point \( v_{out} \in V_{out} \) as an atomic operation. This means that all input required to compute \( f(v_{out}) \), namely \( S_s(v_{out}) \), needs to reside in internal memory to do the calculation and partial computations are not allowed. Refer to Section 3.8 for a discussion of this assumption.
3.3 results

This work examines the leading term of the non-compulsory I/Os of the s-star stencil. For dimension $d \geq 4$, this work improves the lower bound between a factor of 4 and 6. For the upper bounds and $d \geq 3$, the constant of the leading term of the non-compulsory I/Os is analyzed for the first time, resulting in bounds that match up to a factor of $\frac{d}{\sqrt{d!}}$. Furthermore, in three dimensions, the bounds match up to a factor of $\frac{\sqrt{3}}{2}$ and improve the known results by a factor of $\frac{2\sqrt{3}}{\sqrt{B}}$. In two dimensions, this work provides matching constants for the number of non-compulsory I/Os, thus closing a multiplicative gap of 4. All mentioned gaps have existed since 2002.

We use the following assumptions for the asymptotic analysis. The dimension $d$ is assumed to be fixed. Given $d$, we assume that there is an abstract parameter $n$ governing the size of our problem. In particular, $n$ is the parameter which goes to infinity in the $\mathcal{O}$-notation. All other parameters of the problem, the grid sizes $k_i$ ($1 \leq i \leq d$), the size of the internal memory $M$ and the block size $B$ are going to depend on $n$. Hence, when we write $k_i$ we actually mean $k_i(n)$. The same holds for $M(n)$ and $B(n)$ are all positive, non-decreasing functions. The grid sizes $k_i(n)$ are assumed to be ordered by size, i.e. $k_1(n) \geq k_2(n) \geq \cdots \geq k_d(n)$. Furthermore, we assume $\frac{k_d(n)}{M(n)} \xrightarrow{n \to \infty} \infty$ and a weak tall cache assumption, namely $\frac{M(n)}{B(n)} \xrightarrow{n \to \infty} \infty$. In other words, $M(n) = \mathcal{O}(k_d(n))$ and $B(n) = \omega(M(n))$. We regard everything that grows slower than the leading term of the non-compulsory I/Os as lower order terms. Terms that solely depend on $d$ or $s$ are regarded constant.

Denote by $C_s(k_1, \ldots, k_d)$ the number of simple I/Os to evaluate the $s$-point stencil on $[k_1] \times \cdots \times [k_d]$. Then the following holds in the serial case:

$$C_s(k_1, k_2) = \left(2 + \frac{4s^2}{M} \cdot \begin{cases} 1 + \mathcal{O}\left(\frac{B}{M} + \frac{M}{k_1}\right) \\ 1 - \mathcal{O}\left(\frac{1}{M} + \frac{M}{k_2}\right) \end{cases}\right) \cdot \frac{k_1k_2}{B},$$

$$C_s(k_1, k_2, k_3) = \left(2 + \frac{8}{\sqrt{M}} \cdot \frac{s^{3/2}}{\sqrt{M}} \cdot \begin{cases} \sqrt{2} + \mathcal{O}\left(\frac{\sqrt{B}}{M}\right) \\ 1 - \mathcal{O}\left(\frac{1}{\sqrt{M}} + \frac{\sqrt{M}}{k_3}\right) \end{cases}\right) \cdot \frac{k_1k_2k_3}{B},$$

and
\[
C_s(k_1, \ldots, k_d) = \left(2 + \frac{4s \cdot d^{-\frac{1}{2}} \sqrt{2s} \cdot (d-1)}{d^{\frac{1}{2}} \sqrt{M}} \right) \cdot \left\{ \begin{array}{c}
1 + O\left(\frac{d^{-\frac{1}{2}}}{\sqrt{B}}\right) \\
\frac{d^{-\frac{1}{2}}}{\sqrt{d!}} - O\left(\frac{d^{-\frac{1}{2}}}{\sqrt{M}} + \frac{d^{-\frac{1}{2}}}{\sqrt{M}}\right) \end{array} \right\} \cdot \prod_{i=1}^{d} k_i.
\]

The bounds consist of three parts. The first part is the constant 2 accounting for the compulsory I/Os. The second part is the leading term of the non-compulsory I/Os on which this work focuses. The third part characterizes lower order terms that we do not explore further. The best 3-dimensional upper bound is only proven for \(s \in \{1, 2, 3\}\) but should generalize to arbitrary \(s \in \mathbb{N}\).

Parallelization of both, the lower and upper bounds, to the CREW (concurrent read, exclusive write) parallel external memory model (PEM) [AGNS08] is simple when we work not-in-place and the number \(P\) of processors is of order \(O\left(\frac{1}{M} \prod_{i=1}^{d} k_i\right)\). Then, the complexities are reduced by a factor of \(P\). The lower bound of this chapter is derived for \(B = 1\) and works in the parallel setting just as well: as we assume that stencil evaluations are atomic there are no intermediate results. Hence, we can simulate any parallel algorithm using \(P\) processors on a single processor increasing the total number of I/Os by at most a factor of \(P\).¹

This simulation implies that the lower bound in the parallel setting is by at most a factor of \(P\) weaker then the serial lower bound. For the simulation, simply execute all computations of processor \(p_1\) first, followed by all computations of processor \(p_2\) and so forth until we finish with processor \(p_P\). This serialization requires one modification which does not change the total number of I/Os. As we assume simple I/Os, a processor may need to store a vertex back to external memory such that a processor which is simulated later can access this vertex. Consider any particular vertex \(x\) of the grid and say it is read \(k\) times in the serialized algorithm. The vertex needs to be transferred back to external memory the first \(k-1\) times it is evicted from internal memory. Hence, this vertex causes 1 compulsory read and \(k-1\) non-compulsory reads and \(k-1\) non-compulsory writes. The same holds for the number of I/Os this vertex causes in the parallel version of the algorithm. Just the processor that have to perform the non-compulsory writes change. As in the serial setting, the parallel lower bound can be generalized to arbitrary \(B\) by the simple observation that one I/O operation affects at most \(B\) elements.

Regarding the algorithms: as we are working not-in-place, i.e. on an input and an output copy of the grid, all evaluations of stencils are independent from each other and can hence be done in parallel. Therefore,

¹ Unlike with classical computational complexity (i.e. on PRAM), taking advantage of the combined internal memory of the PEM model of size \(P \cdot M\) enables speedups above \(P\) for certain tasks.
we use the proposed serial algorithms and merely split the computation into $P$ contiguous parts. For instance, the work band list $W$ can be split in parts that contain an equal number of work bands and each processor evaluates the evaluation bands corresponding to its part of the work band list. The only additional non-compulsory I/Os are used to initially fill the local memory. Assuming $P = \Theta\left(\frac{1}{M} \prod_{i=1}^{d-1} k_i\right)$, this is a lower order term, namely the one that we analyze as the difference between the torus and the grid.

3.4 related work

Refer to Chapter 2 for the references regarding the cache-aware external memory model and the discussion that this and similar models are limited to measure memory transfer in an idealized scenario while disregarding many effects (associativity, cache replacement strategy, cache allocation strategy, etc.) of actual systems.

The pathwidth of graphs was studied intensely in the series of papers on graph minors by Robertson and Seymour [RS83]. Pathwidth can also be modeled by a robber-cop game [ST93]. For more details about pathwidth and treewidth refer to Bodlaender’s survey [Bod98]. Pathwidth is of interest to us (see Section 3.5.4) as evaluating star-stencils on a graph has to cause non-compulsory I/Os if the pathwidth is at least $M$ (see Lemma 3.6). This enables us to split the algorithm into rounds by its number of non-compulsory I/Os and apply an isoperimetric result to each of these rounds.

The idea of splitting an algorithm into rounds and applying a sort of isoperimetric results goes back to Hong and Kung [HK81]. Hong and Kung use dominator sets to determine how much input has to be loaded to compute a certain round. With that approach they derive the first I/O bounds for products of graphs. In particular, they proof a lower bound of $\Theta\left(\frac{1}{n-M} \cdot \prod_{i=1}^{n} k_i\right)$ I/Os for the graph that is the product of paths. This graph, however, differs significantly from the setup we examine as the product of paths has only one single input and one single output vertex and hence the compulsory I/Os are negligible. Using similar techniques Hong and Kung also derive bounds for problems like the Fast-Fourier-Transform (FFT) and matrix-matrix multiplication.

Hong and Kungs lower bound for matrix-matrix multiplication is extended to a distributed memory setup by Irony et al. [ITT04]. With this work as starting point, a series of papers studies the relation between communication costs of algorithms and expansion properties of the underlying computation graphs for various problems from numerical linear algebra. Problems include Strassen’s matrix-matrix multiplication [BDHS14, BDHS12, BDH¹+12], sparse random matrix-matrix multiplication [BBD¹+13], triangular substitution, Gaussian elimination,
Krylov subspace methods, LU factorization [SCKD14], the Gram-Schmidt algorithm, Cholesky factorization, LDLT-factorization, QR-factorization, eigenvalue and singular value algorithms [BDHS11] and in general programs that reference arrays [CDK+13]. For further references, refer to these papers and the references therein. This research includes lower bounds and also examines trade-offs between local computation and the required communication. The studies focus on the asymptotic complexity of the problem and are often limited to $B = 1$. The expansion properties of the computation graph are closely related to the dominator set of Hong and Kung and the isoperimetric inequality of Bollobás and Leader [BL90] we use to derive our bounds for stencil computations. By carefully adapting the isoperimetric inequality we are able to analyze star-stencil computations past the leading term of the compulsory I/Os and prove a lower bound for the constant of the leading term of the non-compulsory I/Os.

As stencil computations can be regarded as multiplying a sparse matrix with a vector, the I/O complexity of this problem is of particular interest for this chapter. The complexities of sparse matrix-vector multiplication have been derived for sparse random matrices of varying densities in the serial setting [BBF+07, BBF+10] and also in the parallel setting [Gre12]. Furthermore, the complexity of multiplying the same sparse, random matrix with several vectors at once [GJ10] and the complexity of computing the product of two sparse random matrices [PS14] have been analyzed. As one stencil gives rise to one particular sparse matrix, we can exploit the structure of our problem and hence perform better than these bounds which state the worst case complexities over all sparse random matrices of a certain density.

The I/O complexity of the 1-star stencil has already been studied independently by Frumkin and Wijngaart [FVdW02] and Leopold [Leo02b, Leo02a, Leo02c] for arbitrary $B$. Both lines of work examine the leading term of the non-compulsory I/Os but do not analyze it with the precision presented in this chapter. The different results for the leading term of the non-compulsory I/Os are given in Table 3.1 and have to be multiplied by the number of vertices $\prod_{i=1}^{n} k_i$. Frumkin and Wijngaart consider arbitrary dimensions but focus on the asymptotic behavior of the non-compulsory I/Os. The lower bound uses an isoperimetric argument similar to the one presented in this article but does not exploit its full strength. We improve these results by a factor between 4 and 6. The upper bound focuses on the asymptotic behavior and is an existence results. Leopold focuses on the two and three dimensional cases. Her lower bounds exploit a weak isoperimetric result [Leo02b, Leo02c] which we improve by a factor of 2 and $\frac{4}{\sqrt{3}}$ for two respective three dimensions. The upper bounds discuss row and column layouts. By using a data layout suited for our algorithms we decrease the upper bounds by $\frac{1}{2}$ and $\frac{2}{3\sqrt{3}}$ for two and three dimensions. Leopold also discusses two
3.4 Related Work

<table>
<thead>
<tr>
<th>Lower Bounds</th>
<th>Presented Result</th>
<th>Frumkin &amp; Wijngaart</th>
<th>Leopold</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low. Bnd. 2D</td>
<td>$\frac{4}{B\sqrt{M}}$</td>
<td>$\frac{8}{9\sqrt{B\cdot M}}$</td>
<td>$\frac{2}{B\sqrt{M}}$</td>
<td>2</td>
</tr>
<tr>
<td>Low. Bnd. 3D</td>
<td>$\sqrt{\frac{3}{B\cdot \sqrt{M}}}$</td>
<td>$\frac{2}{\sqrt{3 \cdot B \cdot \sqrt{M}}}$</td>
<td>$\frac{2}{B \cdot \sqrt{M}}$</td>
<td>$\frac{4}{\sqrt{3}}$</td>
</tr>
<tr>
<td>Low. Bnd. dD</td>
<td>$\frac{d}{B \cdot \sqrt{M}}$, $\frac{d}{d - \sqrt{3 \cdot (d - 1)}}$, $\frac{d}{d - \sqrt{3 \cdot (d - 1)}}$</td>
<td>$\frac{d}{d - \sqrt{6 \cdot (d - 1)}}$, $\frac{d}{d - \sqrt{6 \cdot (d - 1)}}$</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Upper Bounds</th>
<th>Presented Result</th>
<th>Frumkin &amp; Wijngaart</th>
<th>Leopold</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upp. Bnd. 2D</td>
<td>$\frac{4}{B\sqrt{M}}$</td>
<td>$O\left(\frac{1}{\sqrt{M}}\right)$</td>
<td>$\frac{8}{B\sqrt{M}}$</td>
<td>2</td>
</tr>
<tr>
<td>Upp. Bnd. 3D</td>
<td>$\frac{8\sqrt{3}}{B \cdot \sqrt{M}}$</td>
<td>$O\left(\frac{1}{\sqrt{M}}\right)$</td>
<td>$\frac{4\sqrt{3}}{B \cdot \sqrt{M}}$</td>
<td>$\frac{3}{2} \sqrt{B}$</td>
</tr>
<tr>
<td>Upp. Bnd. dD</td>
<td>$\frac{d - 1}{B \cdot \sqrt{M}}$, $\frac{d - 1}{d - \sqrt{3 \cdot (d - 1)}}$</td>
<td>$O\left(\frac{1}{d - \sqrt{M}}\right)$</td>
<td>n.a.</td>
<td>$O\left(\frac{1}{B}\right)$</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of the bounds for the leading term of the non-compulsory I/Os for the $1$-star stencil ($s = 1$). All to be multiplied with the number of grid points $\prod_{i=1}^{d} k_i$. The best presented result as well as the previously known best result, upon which we improve, are bold.

Spatial and one temporal dimension [Leo02a], which is out of the scope of this chapter (also see Section 3.8).

The research on optimizing stencil computations, mostly in two and three dimensions, on modern computer architectures is vast and ongoing. As stencil computations are not compute intensive, this research focuses on improving the I/O behavior of the algorithms. All known algorithms work on either the standard row- and column-major layout and do not adapt the data layout to the structure of the stencil as done in this chapter. Typically, the number of cache misses is reduced by tiling. Tiling is typically done in either the spatial dimensions alone, increasing spatial locality, or in the spatial and time dimensions, increasing spatial as well as temporal locality.

It was observed that spatial tiling alone has less and less effects due to a refined memory hierarchy and may actually interfere with prefetching techniques [KHO+05, DKW+09]. Furthermore, standard spatial blocking in 3 dimensions is difficult as the blocks would need to be very small to fit in memory [RT00]. Hence, Rivera and Tseng suggest to block only the 2 least significant dimensions reducing the dimensionality of the blocks by 1. This approach is very similar to the one presented in this chapter moving $d - 1$ dimensional sweep shapes through the grid. The results of this chapter also suggest that spatial blocking alone can decrease runtime by just a small fraction as spatial blocking addresses
the leading term of the non-compulsory I/Os but not the dominating term of compulsory I/Os.

Optimizing stencil computations with a time step changes the game. In a time step setup the compulsory I/Os no longer dominate and hence increasing temporal (and spatial) locality can speed up the code significantly. Keep in mind, however, that merging time steps is not applicable in many application domains [KHO+05] and if computations should be performed between time steps [KDW+06, DKW+09]. Hence, improving spatial locality alone is the more general task and always applicable.

Time skewing schemes reorder the computations to enable parallelism and to reduce synchronization points. The first time skewing schemes were derived by Wolfe [Wol89], Song & Li [SL99] and Wonnacott [Won00]. More recently, wavefront approaches have been implemented for multicore chips with shared cache [WHZ+09, TWH11]. In these approaches the different processor of the multicore chip update successive temporal wavefronts of the spatial block improving temporal locality. Also, special hyperplane cuts to enable concurrent startup of the tiles have been derived [BPB12].

Cache-oblivious algorithms for stencil computations using trapezoidal space-time cuts have been derived and analyzed for $B = 1$ [FS05] as well as implemented [FS07, ZWN+08]. The complexity of the cache-oblivious algorithm proposed by Frigo and Strumpen matches the lower bound of Hong and Kung [HK81] asymptotically. Also, cache-oblivious algorithms based on space filling curves and stacks have been derived and implemented [MWZ06, GMPZ06]. As space filling curves can model hierarchical data structures with no memory overhead, they are also very suitable for multigrid methods which are among the most efficient PDE solvers. It was observed that cache-aware algorithms typically outperform their cache-oblivious counterparts for stencil computations [KDW+06, DKW+09]. Furthermore, it was observed that cache-oblivious approaches may increase runtime although they decrease memory traffic.

The literature also includes work on compiler optimization [TCK+11, HVF+13] and auto-tuning. General autotuning of sparse-matrices is provided by the "optimized sparse kernel interface" (OSKI) [VDY05]. For stencil computations, automatic time skewing schemes [LS04], tiling strategies for parallel startup and execution [KBB+07], parallelization strategies [KCO+10] exist. Stencil autotuners also exist for GPUs [CSB11]. Datta’s autotuner applies a wide range of optimizations including problem decomposition techniques, data allocation schemes, bandwidth optimizations and in-core optimizations [Dat09, DMV+08]. The autotuners are complemented by predictive models [RYQ11, SF13] which can guide the optimizations. The autotuners can build upon the polyhedron model\textsuperscript{2} which provides an abstract framework to represent loop programs as

\textsuperscript{2} The polyhedron model is also called polyhedral model or polytope model.
computation graphs. Applying techniques from linear programming allows to reorder this computation graph to enhance parallel execution, minimize the number of synchronization points and optimize performance. Refer to [FL11] for a recap of the model which was stimulated from the architecture [KMW67] as well as the software [Lam74] community.

Bisseling’s survey [Bis04] describes tiling techniques very similar to the ones presented in this chapter. In 2 dimensions, he argues that tiling the space with diamonds ($f^1$ balls) has a better surface to volume ratio than tiling the space with squares. In 3 dimensions, he proposes to tile the space with a truncated octahedron to improve the surface to volume ratio over that of cubes. We prove that the diamond tiling is in fact optimal in 2 dimensions and provide a lower bound (and algorithm) for the 3 dimensional case.

All mentioned stencil algorithms of the literature work on the standard row or column major data layouts. In this chapter we present algorithms that work on non-standard layouts to reduce the memory traffic. While reordering the data to a specific layout may not be worthwhile for a single stencil sweep, reordering should pay off when the stencil is applied repeatedly. In particular, we show that non-standard data layouts are crucial to optimize memory traffic.

3.5 Lower bounds

The lower bound is derived by splitting an arbitrary algorithm into rounds of a certain number of non-compulsory I/Os and applying an isoperimetric result combined with a pathwidth argument to each round. The lower bound is first deduced assuming that an I/O operation accesses one element ($B = 1$) and is then generalized for arbitrary $B$. This section first introduces some notation, mainly from [BL90], to then state the required isoperimetric result. Thereafter the isoperimetric sets are examined and results concerning pathwidth summarized. These findings are then merged to derive the lower bound.

3.5.1 Notation: Fractional Sets, Boundary and Core

The notation necessary to prove and apply the isoperimetric result includes fractional systems, the notion of weight, boundary and interior of these systems and fractional balls as special systems. A fractional system or simply system $f$ is a function from $\mathbb{Z}_k^d$ or $\mathbb{Z}^d$ to the unit interval $[0, 1]$. For $f : \mathbb{Z}^d \to [0, 1]$ the function can take non-zero values only for a finite number of grid points. The weight $w$ of a system $f$ is

$$w(f) = \sum_{x \in \mathbb{Z}_k^d} f(x) \quad \text{or} \quad w(f) = \sum_{x \in \mathbb{Z}^d} f(x)$$
according to the domain of $f$. A fractional system $f$ on $\mathbb{Z}_k^d$ or $\mathbb{Z}^d$ is therefore a generalization of a subset $S$ of $\mathbb{Z}_k^d$ or $\mathbb{Z}^d$ respectively. If a fractional systems $f$ takes just the values 0 and 1, then $f$ is naturally identified with the set $S = f^{-1}(1)$ and the weight $w(f)$ is the cardinality of $S$.

The closure $\partial f$ of a system $f$ is given by

$$\partial f(x) = \begin{cases} 
1, & f(x) > 0 \\
\max_{||x-y||_1=1} \{f(y)\}, & f(x) = 0
\end{cases}$$

Similar to the closure we define the inner core $\Delta f$ of $f$ by

$$\Delta f(x) = \begin{cases} 
0, & f(x) < 1 \\
\min_{||x-y||_1=1} \{f(y)\}, & f(x) = 1
\end{cases}$$

and the inner-s-core by applying the operator repeatedly, $\Delta_s f = \underbrace{\Delta \ldots \Delta}_s f$.

This is now used to define the inner-s-boundary by

$$\Gamma_s f(x) = f(x) - \Delta_s f(x).$$

The fractional $\ell^1$–ball $b_y^{(r, \alpha)}$ of radius $r \in \mathbb{N}_0$, $0 \leq r \leq \frac{k}{2}$, surplus $\alpha \in (0, 1)$ and center $y \in \mathbb{Z}_k^d$ is defined as

$$b_y^{(r, \alpha)}(x) := \begin{cases} 
1, & ||x-y||_1 \leq r \\
\alpha, & ||x-y||_1 = r + 1 \\
0, & ||x-y||_1 > r + 1
\end{cases}$$

For $0 \leq v \leq k^d$ we also use the notation $b_y^v$ which describes the unique ball of weight $v$ and center $y$. For the isoperimetric inequalities the centers of the balls are irrelevant and hence we omit the subscript $y$ when it is not needed.

3.5.2 Isoperimetric Result

The goal of this section is to prove the isoperimetric result given in Theorem 3.2. An isoperimetric inequality states how many vertices can be enclosed by a fixed number of boundary vertices. The optimal sets in this sense are called isoperimetric sets and, as proven by Bollobás and Leader [BL90], the isoperimetric sets in $\mathbb{Z}_k^d$ are (fractional) $\ell^1$–balls. Precisely, Bollobás and Leader have proven that $\ell^1$ balls have the smallest closure of all systems of the same weight.

---

3 It is known that the isoperimetric sets in the continuous domains $\mathbb{R}^d$ are $\ell^2$ balls.
Theorem 3.1 (An isoperimetric inequality on the discrete torus).
For $k \geq 2$ and $k$ even, let $f$ be a fractional system on $\mathbb{Z}_k^d$. Then the following holds:

$$w(\partial f) \geq w\left(\partial b^{w(f)}\right).$$

Proof. The result has been proven by Bollobás and Leader [BL90] as Theorem 4.

We need a version of this result which enables us to bound the number of interior vertices given the number of inner-boundary vertices. This differs in two aspects from the above theorem: First, we want to look at the boundary as part of the set, the inner-boundary, and not add it in addition like in the closure. Second, we need to have a result for systems of all weights but bounded inner-boundary. This will make it necessary to translate Theorem 3.1 to the infinite grid $\mathbb{Z}^d$ where the boundary of the balls is growing strictly monotonic. The desired result reads:

Theorem 3.2 (The boundary bounds the core on $\mathbb{Z}^d$).
Let $s \in \mathbb{N}$ and $f$ be a fractional system on $\mathbb{Z}^d$. For $v \in \mathbb{R}_0^+$ the following holds:

$$(w(\Gamma_{2s} f) \leq w(\Gamma_{2s} b^v)) \Rightarrow (w(\Delta_s f) \leq w(\Delta_s b^v)). \quad (3.1)$$

We first prove two lemmata.

Lemma 3.3. For a fractional system $f$ on $\mathbb{Z}^d$ the following inequality holds:

$$\partial(\Delta f)(x) \leq f(x).$$

Proof. The claim is proven by examining the three different cases carefully. If $f(x) = 0$ it follows that $\partial(\Delta f)(x) = 0$ as well since all neighbors of $x$ are set to 0 by the $\Delta$-operator. When $0 < f(x) < 1$, $\Delta f(x) = 0$ and for all $y$ such that $||x - y|| = 1$ we have $\Delta f(y) \leq f(x)$ and hence $\partial(\Delta f)(x) \leq f(x)$. When $f(x) = 1$ the claim holds trivially.

The second lemma states that balls have the largest inner-core of all systems of the same weight.

Lemma 3.4 (A version of the isoperimetric inequality).
For even $k$, $s \in \mathbb{N}$ and all fractional systems $f$ on $\mathbb{Z}^d$ it holds that

$$w(\Gamma_s f) \geq w\left(\Gamma_s b^{w(f)}\right)$$

which is by definition equivalent to

$$w(\Delta_s f) \leq w\left(\Delta_s b^{w(f)}\right). \quad (3.2)$$

Proof. The claim is proven by induction over $s$. First, consider the case $s = 1$. If $w(f) \leq 1$ then $w(\Gamma f) = w(f)$ and $w(\Delta f) = 0$ such that the claim
holds. Assume there exists some fractional system \( f \) with \( w(f) > 1 \) such that
\[
w(\Gamma f) < w\left( \Gamma b^{w(f)} \right) \quad \text{and hence} \quad w(\Delta f) > w\left( \Delta b^{w(f)} \right).
\]
By the latter and the strict monotonicity of \( w\left( \partial b^{(s)} \right) \) we get
\[
w\left( \partial b^{w(\Delta f)} \right) > w\left( \partial b^{w(\Delta b^{w(f)})} \right).
\]
To simplify the right hand side we use that the inner core of a ball is itself a ball and hence we can discard building the ball of it.
\[
w\left( \partial b^{w(\Delta b^{w(f)})} \right) = w\left( \partial \Delta b^{w(f)} \right).
\]
For a ball with \( w(f) > 1 \) the closure of the inner core is pointwise equal to the ball itself. Furthermore we employ Lemma 3.3.
\[
w\left( \partial \Delta b^{w(f)} \right) = w\left( b^{w(f)} \right) = w(f) \geq w(\partial \Delta f).
\]
Reading this sequence of inequalities altogether yields
\[
w\left( \partial b^{w(\Delta f)} \right) > w(\partial \Delta f).
\]
Since \( f \) takes just a finite number of non-zero values, we can find \( k \) such that all non-zero values of \( f \) are in the grid \( \{-k, \ldots, k\}^n \) and we can embed \( f \) in the torus \( \mathbb{Z}^n_{2k+3} \) such that no points of \( f \) touch were the grid is closed to a torus. Therefore we can transfer the counterexample to the torus where it contradicts Theorem 3.1 for \( \Delta f \) as fractional system and proves the claim for \( s = 1 \).

Let us now prove the claim for \( s \) assuming it holds for \( s - 1 \). Using the induction assumption for \( \Delta f \) we arrive at
\[
w(\Delta_s f) = w(\Delta_{s-1}\Delta f) \leq w\left( \Delta_{s-1} b^{w(\Delta f)} \right).
\]
Noting that \( b^\prime, \Delta b^\prime \) and \( \Delta_{s-1} b^\prime \) are pointwise monotonically increasing yields that \( w(\Delta_{s-1} b^\prime) \) is monotonically increasing. Hence we can apply the result proven for \( s = 1 \) to yield
\[
w\left( \Delta_{s-1} b^{w(\Delta f)} \right) \leq w\left( \Delta_{s-1} b^{w(\Delta b^{w(f)})} \right).
\]
But the inner core of a ball is a ball itself, so we can discard building the ball of it and this simplifies to the required result
\[
w\left( \Delta_{s-1} b^{w(\Delta b^{w(f)})} \right) = w\left( \Delta_{s-1} \Delta b^{w(f)} \right) = w\left( \Delta_s b^{w(f)} \right).
\]
Since the weight of the inner-s-core of a ball is monotonically increasing with the weight of the ball, this result can be used to deduce the implication

\[(w(f) \leq v) \Rightarrow (w(\Delta_s f) \leq w(\Delta_s b^v))\].

Nevertheless, we run into problems when bounding the weight of a ball given that its inner-boundary is bounded. The inner-boundary of balls is only monotonically increasing until \(v \approx \frac{k^n}{2}\) and thereafter monotonically decreasing. To overcome this problem, Theorem 3.2 transfers the results to the infinite grid, where the inner-s-boundary of balls is monotonically increasing with respect to the weight of the ball.

**Proof of Theorem 3.2.** The proof is split into two parts. First, we prove \((w(\Gamma_2 s f) \leq w(\Gamma_2 s b^v)) \Rightarrow (w(f) \leq v)\) by contraposition. I.e., we first show

\[(w(\Gamma_2 s f) > w(\Gamma_2 s b^v)) \iff (w(f) > v)\].

From Lemma 3.4, namely \(w(\Gamma_s f) \geq w(\Gamma_s b^{w(f)})\), and the observation that the weight \(w(\Gamma_s b^v)\) is strictly monotonically increasing on \(\mathbb{Z}^n\) with respect to \(v\), it follows that

\[w(\Gamma_1 f) \geq w(\Gamma_s b^{w(f)}) > w(\Gamma_s b^v)\].

Since \(s\) was arbitrary, it also follows that \(w(\Gamma_2 s f) > w(\Gamma_2 s b^v)\) which establishes the first part.

Employing Lemma 3.4 again and noting that \(w(\Delta_s b^v)\) is monotonically increasing with respect to \(v\) yields

\[(w(f) \leq v) \Rightarrow (w(\Delta_s f) \leq w(\Delta_s b^v))\].

This completes the proof. \(\square\)

### 3.5.3 Size of the \(\ell^1\)–Ball and its Boundary

This section derives the asymptotic expansion for the number of vertices of a ball and its inner-boundary in \(\mathbb{Z}^d\) with respect to the radius \(r\),

\[w\left(b^{(r,0)}\right) = \frac{2^d}{d!} \cdot r^d + \Theta\left(r^{d-1}\right)\] \hspace{1cm} (3.3)

and

\[w\left(\Gamma_1 b^{(r,0)}\right) = \frac{2^d}{(d-1)!} \cdot (r-1)^{d-1} + \Theta\left(r^{d-2}\right)\]. \hspace{1cm} (3.4)
Both lower order terms are non-negative and the dimension $d$ is assumed to be constant. As long as the sides of the torus or grid are big enough, $k \geq 2(r + 1)$, the formulas apply there also.

We derive these formulas by recursion over the dimensions. Hence, it is useful to introduce the notation $b_d^{(r,0)}$ for the ball of radius $r$ in $d$ dimensions. The $\ell^1$-ball of dimension $d$ consists of smaller balls of one dimension less, namely the level sets in the new dimension. Therefore,

$$w(b_d^{(r,0)}) = w(b_d^{(r-1,0)}) + 2 \cdot \sum_{l=0}^{r-1} w(b_{d-1}^{(l,0)}) . \tag{3.5}$$

Another simple fact is $b_d^{(r,0)} = b_d^{(r-1,0)} + \Gamma b_d^{(r,0)}$. In combination with Equation 3.5 this yields

$$w(\Gamma b_d^{(r,0)}) = w(b_d^{(r,0)}) + w(b_d^{(r-1,0)}) . \tag{3.6}$$

Since $w(\Gamma b_d^{(0,0)}) = 1$ for all $d \in \mathbb{N}$ and $w(\Gamma b_1^{(r,0)}) = 2$ for $r \geq 1$ the weight of the one-dimensional balls is given by

$$w(b_1^{(r,0)}) = 2r + 1 . \tag{3.7}$$

Equation 3.5 and Equation 3.6 yield that $w(b_d^{(r,0)})$ and $w(\Gamma b_d^{(r,0)})$ are polynomials in $r$ of degree $d$ and $d - 1$ respectively. So they can be written as

$$w(b_d^{(r,0)}) = \sum_{i=0}^{d} \alpha_{(d,i)} \cdot r^i \quad \text{and} \quad w(\Gamma b_d^{(r,0)}) = \sum_{i=0}^{d-1} \beta_{(d,i)} \cdot r^i .$$

First, let us derive an upper bound for $\alpha_{(d,d)}$, the leading term of $w(b_d^{(r,0)})$:

$$w(b_d^{(r,0)}) = w(b_d^{(r-1,0)}) + 2 \sum_{l=0}^{r-1} w(b_{d-1}^{(l,0)}) =$$

$$= O(r^{d-1}) + 2 \sum_{l=0}^{r-1} \sum_{i=0}^{d-1} \alpha_{(d-1,i)} \cdot l^i =$$

$$= O(r^{d-1}) + 2 \sum_{i=0}^{d-1} \left( \alpha_{(d-1,i)} \sum_{l=0}^{r-1} l^i \right) \leq$$

$$\leq O(r^{d-1}) + 2 \sum_{i=0}^{d-1} \left( \alpha_{(d-1,i)} \int_0^r l^i \, dl \right) =$$
\[= \Theta(r^{d-1}) + 2 \sum_{i=0}^{d-1} \left( \alpha_{(d-1, i)} \frac{r^{i+1}}{i+1} \right) =
\]
\[= 2 \alpha_{(d-1, d-1)} \frac{r^d}{d} + \Theta(r^{d-1}).\]

Comparing the coefficient of the leading terms yields the recursion
\[\alpha_{(d, d)} \leq \frac{2d}{d!} \alpha_{(d-1, d-1)}.\]

The recursion stops with Equation 3.7, namely \(\alpha_{(1, 1)} = 2\). Hence we get
\[\alpha_{(d, d)} \leq \frac{2d}{d!} \quad \text{and} \quad w\left(\Gamma b_d^{(r,0)}\right) \leq \frac{2d}{d!} \cdot r^d + \Theta(r^{d-1}).\]

In consequence, Equation 3.6 yields an upper bound for the boundary and its leading term:
\[\beta_{(d, d-1)} \leq \frac{2d}{(d-1)!} \quad \text{and} \quad w\left(\Gamma b_d^{(r,0)}\right) \leq \frac{2d}{(d-1)!} \cdot (r-1)^{d-1} + \Theta(r^{d-2}).\]

Next, we are going to show that the coefficients \(\alpha_{(d, d)}\) and \(\beta_{(d, d-1)}\) take these values and that the lower order terms \(\Theta(r^{d-1})\) (for the ball) and \(\Theta(r^{d-2})\) (for the boundary of the ball) are non-negative. Therefore, let us prove a lower bound for number of points of \(b_d^{(r,0)}\). Instead of counting the number of points of \(b_d^{(r,0)}\), consider the following d-dimensional body: in \(\mathbb{R}^d\), replace each point \(x \in b_d^{(r,0)}\) with a d-dimensional cube of side length 1 and center \(x\). If \(y_i\) denotes the i-th component of \(y\), then the cube corresponding to the point \(x\) is given by
\[
\left\{ y \in \mathbb{R}^d : y_i - x_i \in \left[ -\frac{1}{2}, \frac{1}{2} \right] \forall i \in \{1, \ldots, d\} \right\}.
\]

Each cube has volume, i.e., measure, of 1 in the d-dimensional Lebesgue measure. As the points of \(x \in b_d^{(r,0)}\) have integer coefficients, the cubes do not overlap (to be precise, their overlap has measure 0). Therefore, the volume of the constructed body is precisely the number of points in \(b_d^{(r,0)}\).

To lower bound the volume of the body, consider the d-dimensional \(\ell^1\) ball of radius \(r\) as d-dimensional polytope in \(\mathbb{R}^d\). For \(r = 1\), this polytope is also called cross polytope and has volume \(2^d/(d!)\). For arbitrary radius \(r\), the volume of the \(\ell^1\) ball of radius \(r\) is given by
\[\frac{2^d}{d!} \cdot r^d\]
and follows from scaling each dimension of the cross polytope by a factor of \( r \). As the body we constructed started with \( b_d^{(r,0)} \), i.e., the integral points of the \( d \)-dimensional \( \ell^1 \) ball of radius \( r \), the constructed body covers the \( d \)-dimensional \( \ell^1 \)-ball of radius \( r \). Therefore, the constructed body has at least volume \( 2^d/(d!) \cdot r^d \). It follows that \( \alpha_{(d,d)} = 2^d/(d!) \) and that the lower order term in Equation 3.3 is non-negative.

For the boundary, it then follows from Equation 3.6 that

\[
\omega \left( \Gamma b_d^{(r,0)} \right) = \frac{2^d}{(d-1)!} \cdot (r-1)^{d-1} + O\left(r^{d-2}\right)
\]

with a non-negative term \( O(r^{d-2}) \). In particular, \( \beta_{(d,d-1)} = 2^d/(d-1)! \).

### 3.5.4 Pathwidth

We employ pathwidth [RS83] to ensure that we are working on the “inside” of the torus and can treat it like the infinite grid which enables us to apply Theorem 3.2.

**Definition 3.5 (Pathwidth [RS83]).** Let \( G = (V, E) \) be a graph. A sequence \((X_1, X_2, \ldots, X_r)\) of subsets of the vertices \( V \) of \( G \) is a path decomposition of \( G \) if

1. \( \bigcup_{1 \leq i \leq r} X_i = V \).
2. for all edges \((v, w) \in E\) there exists an \( i \in \{1, \ldots, r\} \) such that \( v \in X_i \) and \( w \in X_i \).
3. for all \( i, j, k \) such that \( 1 \leq i \leq j \leq k \leq r \) it holds that \( X_i \cap X_k \subseteq X_j \).

The subsets \( X_i \) are called bags of the path decomposition. The width of a path decomposition \((X_1, X_2, \ldots, X_r)\) is \( \max_{1 \leq i \leq r} |X_i| - 1 \). The width of a graph \( G \) is the minimum width over all possible path decompositions of \( G \).

Condition (3) implies that a vertex can only be in a consecutive block of bags and not reappear after it has been removed from a bag once.

**Lemma 3.6.** Let \( G \) be a graph. Denote by \( M \) the size of the internal memory. If \( \text{pathwidth}(G) \geq M \), then any algorithm evaluating the \( s \)-star on \( G \) has to cause non-compulsory I/Os.

**Proof.** We will prove the contraposition of the claim: If there exists an algorithm evaluating the \( s \)-star stencil on \( G \) with only compulsory I/Os then \( \text{pathwidth}(G) < M \).

If we can evaluate the \( s \)-star stencil on \( G \) with internal memory of size \( M \) and without loading a vertex twice this immediately induces a path decomposition with bags of size at most \( M \). The bags are the different sets of elements the internal memory is containing at different stages of the algorithm and hence \( \text{pathwidth}(G) \leq M - 1 \). \( \square \)
Lemma 3.7. Evaluating the s-star stencil on a two dimensional grid or torus with \( \min(k_1, k_2) \geq M \) has to cause non-compulsory I/Os.

Proof. By Corollary 89 of [Bod98], the two dimensional grid \([k_1] \times [k_2] \) has pathwidth \( \min(k_1, k_2) \). Hence, the claim follows from Lemma 3.6.\[
\]

Pathwidth can also be modeled by a robber and cop game [ST93]. The robber and cop game is played on an arbitrary undirected graph like the grid or the torus. Initially \( p \) cops are placed on the vertices of the graph and afterwards the robber chooses its initial position. The robber is visible to the cops during the game and the game proceeds in rounds. First the cops announce were they want to be placed in the next round. Then every cop that wants to move boards a helicopter. While the cops are moving in the air, the robber can move to an arbitrary vertex of the graph if he can reach it without running into a cop. Thereafter the cops land and the robber escapes in that round if no cop lands on the vertex the robber is standing on. The game then continues with the next round. If there is a strategy so that the robber is able to escape the cops for an infinite number of rounds we say that the robber wins.

The following implication holds [ST93]: When the robber cop game is played with \( p \) cops on a graph \( G \) and if there is a strategy so that the robber wins, \( G \) has to have pathwidth bigger than \( p - 1 \).

Lemma 3.8. If the subgraph \( H \) of a two dimensional grid or torus consists of \( p + 1 \) complete rows and complete columns, then \( \text{pathwidth}(H) \geq p \).

Proof. To prove the claim we give a strategy in the robber and cop game such that the robber wins against \( p \) cops for any strategy the cops have. Since there are \( p + 1 \) complete rows and columns in \( H \), the robber is free to start in a row which is empty, after the initial placements of the cops. When the cops announce their move, there will be a free column in the next configuration. Since the robber is in a free row, it can move to this column which is free in the next configuration. The game now proceeds with rows and columns interchanged. The robber always escapes from a free row to a free column and vice versa.\[
\]

Lemma 3.9. Let \( M \) be the size of the internal memory. If the subgraph \( H \) of a two dimensional grid or torus consists of \( M + 1 \) complete rows and complete columns any algorithm evaluating the s-star stencil on \( H \) has to cause non-compulsory I/Os.

Proof. The claim follows from combining Lemma 3.8 with Lemma 3.6.\[
\]
3.5.5 Splitting into Rounds and Deducing the Lower Bound

To derive the lower bound it is left to describe how to split an algorithm into rounds. Therefore assume an arbitrary algorithm evaluating the s-star stencil on \( \mathbb{Z}_{k_1} \times \ldots \times \mathbb{Z}_{k_d} \) is given. As \( k_i(n) = \Omega(M(n)) \) \( \forall i \) it follows that \( \min\{k_1,k_2\} \geq M \) for almost all \( n \). In these cases, the algorithm causes non-compulsory I/O operations by Lemma 3.7. We can count these operation and split the algorithm into rounds of \( c \) non-compulsory I/Os. \( c \) denotes the round length and hence all rounds except the last one cause \( c \) non-compulsory I/Os. This approach is similar to the idea presented by Hong and Kung [HK81] and therefore we call the rounds Hong-Kung rounds.

To apply the isoperimetric inequality we need to establish a link between the inner-core, the inner-boundary and the rounds. Choose one of the Hong-Kung rounds and denote with \( S \) the set of vertices which are in internal memory at some point of this round. Let \( \text{Transfer}(S) \) be the transfer vertices of \( S \), i.e. vertices which are also present in internal memory during other rounds. Precisely, a vertex is a transfer vertex if at least one of four cases applies:

- The vertex is transferred from the previous to the current round by residing in internal memory at the beginning of the current round.
- The vertex has been written back to external memory in a preceding round and is read again in the current round.
- The vertex is written from internal to external memory in the current round to be read again in a subsequent round.
- The vertex is transferred from the current to the proceeding round by residing in internal memory at the end of the current round.

We denote further \( \text{Eval}(S) \) as the evaluated vertices which are all vertices of \( S \) for which the s-point stencil is evaluated in the current round. The following two observations relate these sets to the inner-core and the inner-boundary:

\[
\Gamma_{2s}(S) \subset \text{Transfer}(S) \tag{3.8}
\]

and

\[
\text{Eval}(S) \subset \Delta_s(S) . \tag{3.9}
\]

A vertex can only be evaluated in a round if all its neighbors within distance \( s \) are in \( S \) as well. \( \Delta_s(S) \) consists of exactly these vertices. Equivalently \( \Gamma_s(S) \) are the vertices which cannot be evaluated in round \( S \). Take any \( x \in \Gamma_s(S) \). All vertices which are within distance \( s \) from \( x \) need to be in the round in which \( x \) is evaluated. Hence they need to be transferred. The set of all vertices of \( S \) within distance \( s \) from any of the vertices
of \( \Gamma_s(S) \) is \( \Gamma_{2s}(S) \). Therefore these vertices are a subset of the transfer vertices.

Furthermore, we can give an upper bound for the number of transfer vertices of a round. At the beginning and at the end of a round there are at most \( M \) vertices in internal memory. Together these account for at most \( 2M \) transfer vertices. The only other way a vertex can be a transfer vertex is that it has been rewritten to external memory in a previous round and is reloaded in the current round or rewritten to external memory in the current round to be reloaded in a subsequent round. So either the reload or write of the vertex causes a non-compulsory I/O. Since there are at most \( c \) non-compulsory I/Os per round, the total number of transfer vertices is at most \( 2M + c \),

\[
    w(\text{Transfer}(S)) \leq 2M + c. \tag{3.10}
\]

We can embed \( S \) in the infinite grid since the torus is assumed to be large. Denote by \( e_i \) the vector of the \( i \)-th unit direction. From \( k_i(n) = \Omega(M(n)) \) \( \forall i \) it follows that

\[
    k_1, k_2 \geq 2M + c + (M + 1) \quad \text{and} \quad k_i \geq 2M + c + 1 \quad \text{for} \quad i \in \{3, \ldots, d\} \quad \text{and almost all} \quad n.
\]

In these cases we know by Equation 3.10 that the vertices of (at least) \( M + 1 \) hyperplanes of normal \( e_1 \), \( M + 1 \) hyperplanes of normal \( e_2 \) and one hyperplane of normal \( e_i \) (\( 3 \leq i \leq d \)) do not belong to \( \text{Transfer}(S) \). The union \( U \) of these hyperplanes forms a connected component in \( \mathbb{Z}_{k_1} \times \cdots \times \mathbb{Z}_{k_d} \). As a connected component \( U \) could either be a subset of \( S \setminus \text{Transfer}(S) \) or disjoint from \( S \). Assume that \( U \subset (S \setminus \text{Transfer}(S)) \). Taking the union of all hyperplanes of normal \( e_1 \) and normal \( e_2 \) and intersecting them with all other hyperplanes results in a subset \( H \subset U \) of a two dimensional torus of at least \( M + 1 \) complete rows and columns. By Lemma 3.9 evaluating the \( s \)-star stencil on \( H \) has to cause non-compulsory I/Os. But evaluating the \( s \)-star stencil for vertices of \( S \setminus \text{Transfer}(S) \) does not cause non-compulsory I/Os by definition. Hence the case \( U \subset (S \setminus \text{Transfer}(S)) \) is not possible and it follows that \( U \) is disjoint from \( S \). Therefore, at least one hyperplane of each normal direction \( e_i \) (\( 1 \leq i \leq d \)) is disjoint from \( S \). Deleting these hyperplanes enables us to embed \( S \) in the infinite grid \( \mathbb{Z}^d \).

Treating \( S \) as a subset of the infinite grid enables us to apply Theorem 3.2 and yields the lower bound. Denote with \( v_0 \) the weight such that

\[
    w(\Gamma_{2s} b^{v_0}) = 2M + c. \tag{3.11}
\]

Combining Equation 3.10 and Equation 3.8 reads

\[
    w(\Gamma_{2s}(S)) \leq w(\text{Transfer}(S)) \leq 2M + c = w(\Gamma_{2s} b^{v_0}).
\]
By Theorem 3.2 and Equation 3.9 it follows that

$$w(\text{Eval}(S)) \leq w(\Delta S) \leq w(\Delta b^{v_0}).$$

Therefore, a lower bound for the evaluation of the $s$-point stencil on $Z_{k_1} \times \cdots \times Z_{k_d}$ is given by

$$\frac{c}{w(\Delta b^{v_0})} \cdot \prod_{i=1}^{d} k_i. \quad (3.12)$$

It is left to determine the round length $c$ that gives the best lower bound. Using the assumption that $s$ is small and constant we simplify Equation 3.11 before solving. Denote $(r_0, \alpha_0)$ the radius and surplus such that $b^{v_0} = b^{(r_0, \alpha_0)}$. Using Equation 3.4, the asymptotic expansion of $w(\Gamma_{2s} b^{v_0})$ is given by

$$w(\Gamma_{2s} b^{v_0}) = \sum_{i=0}^{2s-1} \frac{2^d \cdot (r_0 - 1 - i)^{d-1}}{(d-1)!} + o \left( \frac{r_0^{d-2}}{r_0} \right) = \frac{2s \cdot 2^d}{(d-1)!} \frac{(r_0 - 2s)^{d-1}}{2M} + o \left( \frac{r_0^{d-2}}{r_0} \right). \quad (3.13)$$

Since the lower order term in Equation 3.4 is non-negative, so is the lower order term in Equation 3.13. Therefore, dropping the lower order term before solving Equation 3.13 yields an upper bound for $r_0$ and $v_0$, an upper bound for $w(\Delta b^{v_0})$ and hence a lower bound when these values are used in Equation 3.12. Solving Equation 3.11 without the lower order terms yields

$$r_0 = \frac{d-1}{\sqrt{(d-1)! \cdot \frac{2M + c}{2s \cdot 2^d}}} + 2s. \quad (3.14)$$

The round length $c$ giving the strongest lower bound is chosen by plugging Equation 3.14 into Equation 3.12 and maximizing over $c$ by setting the derivative to 0 and checking that the solution is a maximum. As the round length can be chosen arbitrarily we disregard lower order terms when solving and choose

$$c = 2(d - 1) \cdot M.$$

Using this round length in Equation 3.14, we determine an upper bound for the radius of a ball to be handled in one round as

$$r_0 = \sqrt[2d]{\frac{d! \cdot M}{s}} + 2s.$$
Finally, by plugging this radius into Equation 3.12 and using Equation 3.3 to simplify, the lower bound reads

\[
\frac{2(d-1)M}{\omega(\Delta_s b(r_0, 0))} \cdot \prod_{i=1}^{d} k_i \geq \frac{2(d-1)M}{\omega(b(r_0, 0))} \cdot \prod_{i=1}^{d} k_i =
\]

\[
= \frac{2(d-1)M}{\frac{2^d}{\pi^d} \left( \left( \frac{d-1}{\pi^{d/2} \cdot M} + 2s \right)^d \right) + \mathcal{O}(M)} \cdot \prod_{i=1}^{d} k_i =
\]

\[
= \frac{2(d-1)M}{\frac{2^d}{\pi^d} \left( \left( \frac{d-1}{\pi^{d/2} \cdot M} \right) + \mathcal{O}(M) \right)} \cdot \prod_{i=1}^{d} k_i =
\]

\[
= \frac{2(d-1)M}{\frac{2^d}{\pi^d} \left( \frac{d-1}{\pi^{d/2} \cdot M} \right)^d + \mathcal{O}(M)} \cdot \prod_{i=1}^{d} k_i =
\]

\[
= \left( 4s \cdot \frac{d-1}{\sqrt{2s}} \cdot (d-1) \cdot \frac{1}{d!} \cdot \frac{1}{\sqrt{d-1/M}} \right) \cdot \prod_{i=1}^{d} k_i =
\]

\[
= \left( 4s \cdot \frac{d-1}{\sqrt{2s}} \cdot (d-1) \cdot \frac{1}{d!} \cdot \frac{1}{\sqrt{d-1/M}} + \mathcal{O}(1) \right) \cdot \prod_{i=1}^{d} k_i .
\]

This bound was derived on the torus \( Z_{k_1} \times \cdots \times Z_{k_d} \) and we can apply it to the grid \([k_1] \times \cdots \times [k_d]\) using a reduction.

**Lemma 3.10.** Let \( M \) denote the size of the internal memory. Any algorithm evaluating the \( s \)-point stencil on the grid \([k_1] \times \cdots \times [k_d]\) induces an algorithm evaluating the \( s \)-point stencil on the torus \( Z_{k_1} \times \cdots \times Z_{k_d} \) causing at most \( \mathcal{O} \left( \prod_{i=1}^{d-1} k_i \right) \) additional I/Os.

**Proof.** When the algorithm for the grid is evaluated on the torus, only the vertices close the boundary of the grid have to be treated differently. If a vertex is within \( \ell^1 \) distance \( s-1 \) in a unit direction from a bounding hyperplane of the grid, at most half of the points of the \( s \)-point stencil, corresponding to that unit direction, have to be read and written additionally for this vertex on the torus. Altogether the number of I/Os is at most

\[
\frac{b^{(s,0)}}{2} \cdot s \cdot 2 \cdot \sum_{j=1}^{d} \prod_{i=1}^{d} k_i = \mathcal{O} \left( \prod_{i=1}^{d-1} k_i \right) .
\]
Furthermore, the lower bound can be generalized to arbitrary $B$ by the simple observation that one I/O operation affects at most $B$ elements. Hence, for the grid the total number of I/Os, including the compulsory ones, is

$$\left(2 + \frac{4s \cdot d^{-\frac{1}{2}} \sqrt{2s} \cdot (d-1)}{d^{-\frac{1}{2}} d!} \cdot \frac{1}{d^{-\frac{1}{2}} \sqrt{M} - \Theta \left(\frac{1}{d^{-\frac{1}{2}} \sqrt{M^2} + \frac{1}{k_d}}\right)}\right) \prod_{i=1}^{d} k_i^i B.$$ 

### 3.6 Notation and Algorithmic Framework for the Upper Bounds

Algorithms evaluating the $s$-star stencil on the $d$ dimensional grid $[k_1] \times \cdots \times [k_d]$ are presented in this section. A single time step is considered, i.e. the whole grid is updated once according to the stencil. Hence we limit ourselves to spatial tiling. It has been observed that tiling is limited to spatial tiling in many application domains [KHO+05] and if computations should be performed between time steps [KDW+06, DKW+09].

As we are working not-in-place, we always keep two copies of the grid, one with the input values and one with the output values according to the stencil. Also, we consider simple I/Os in the sense that data items are moved between external and internal memory and hence the original value of a vertex has to be stored back if it is accessed at a later point in time.

All the upper bounds have in common that a sweep shape is moved through the grid in unit shifts in a simple sweep sequence resulting in work bands. This approach is similar to the one employed by Leopold [Leo02b].

The data layout is crucial to the design of a memory efficient algorithm: First, the data layout and the sweep shape have to match. Second, vertices that belong to different sets of work bands, these sets are going to be defined as $k$-intersections, have to be stored together. This section first introduces the necessary definitions and ideas for the Band Algorithms given in Section 3.7 and then states and proves the algorithmic framework to analyze their complexity.

Many of the involved definitions and constructions are necessary as we want to analyze the constant of the leading term of the non-compulsory I/Os and derive the asymptotic behavior of the lower terms. Limiting the analysis to the asymptotic complexity of leading term of the non-compulsory I/Os or disregarding lower order terms would simplify the analysis very much. Furthermore, the Diagonal Band Algorithm and the Hexagonal Band Algorithm, the best algorithms presented in two and three dimensions respectively, employ several different unit shifts which further complicates the analysis. However, this work comes to the conclusions that employing different unit shifts is crucial when matching upper and lower bounds should be designed. Hence, we present and
analyze these algorithm to disclose design issues which are essential for matching bounds.

3.6.1 Notation and Setup for the Upper Bounds

This section first introduces the definitions for sweep sequences, sweep shapes, work bands, evaluation bands and k-intersections. Then, we examine the sweep shapes in more detail defining the size parameter $m$ of a sweep shape and the data layout within a sweep shape. Finally, we reduce the dimensionality of the problem to $d - 1$ by cutting it with hyperplanes. Let us start with the definitions.

The vector of the $i$.th unit direction is denoted by $e_i$. For a vertex $w \in [k_1] \times \cdots \times [k_n]$ or $w \in \mathbb{Z}^d$ the $i$.th component is denoted by $w_i$. Recall the definition of the $s$-star stencil $S_s(w)$ of a vertex $w \in [k_1] \times \cdots \times [k_n]$:

$$S_s(w) = \{v \in [k_1] \times \cdots \times [k_n] : \|v - w\|_1 \leq s\}.$$

A simple sweep sequence or just sweep sequence $X$ of length $k$ is the sequence of the first $k$ unit directions $e_i$ ordered by increasing $i$. We denote by $\delta_i \in X$ the $i$.th element of the simple sweep sequence.

A sweep shape $\mathcal{S}$ is a subset of vertices of the infinite grid $\mathbb{Z}^d$. All considered sweep shapes are the integral points of $d - 1$ dimensional polygons lying in a hyperplane of normal $\sum_{\delta_i \in X} \delta_i$. In two dimensions the sweep shapes are therefore simply line segments. In three dimensions, the employed sweep shapes can be described by squares, diamonds and hexagons. Therefore, you can think of the sweep shapes as convex and $d - 1$ dimensional objects. However, we do not define the notions of convexity and the dimension of a set of points for the discrete setting. The distance of a sweep shape $\mathcal{S}$ from the origin is defined as the $\ell^2$ distance of the hyperplane containing $\mathcal{S}$ from the origin.

For the following discussion and definitions, it is assumed that a simple sweep sequence and a sweep shape have been chosen.

An infinite work band $W^\infty$ is a subset of the infinite grid $\mathbb{Z}^d$. $W^\infty$ results from shifting a sweep shape $\mathcal{S}$ according to the sweep sequence $X$ over and over in positive (or negative) direction. If the end of the sweep sequence is reached, we start over with the first element of the sweep sequence. We say that a sweep shape $\mathcal{S}'$ proceeds (precedes) the sweep shape $\mathcal{S}$ if it results from $\mathcal{S}$ be applying the next (previous) unit shift of the sweep sequence. The infinite work band resulting from $\mathcal{S}$
and $\mathcal{X}$ is given by (assuming that the elements $\delta_i$ of the sweep sequence are indexed from 0 to $|\mathcal{X}| - 1$)

$$W^\infty = \left\{ y \in \mathbb{Z}^d : \exists z \in \mathscr{S}, \exists r \in \mathbb{Z} \text{ such that:} \right\}$$

- if $r \geq 0$: $y = z + \sum_{i=1}^{r} \delta_{((i-1) \mod |\mathcal{X}|)}$
- if $r < 0$: $y = z - \sum_{i=r}^{-1} \delta_{(i \mod |\mathcal{X}|)}$

Each infinite work band $W^\infty$ corresponds to a (finite) work band $W$ containing the vertices of $W^\infty$ that are part of the grid $[k_1] \times \cdots \times [k_d]$, $W = W^\infty \cap ([k_1] \times \cdots \times [k_d])$.

In addition to the sweep sequence and sweep shape, an algorithm is going to be defined by a list of work bands $\mathcal{W}$ which are evaluated one by one. All work bands of one particular algorithm result from the same, possibly shifted, sweep sequence and shape. Different work bands are obtained by shifting the sweep shape to a new start position before applying the sweep sequence.

We associate an evaluation band $E_W$ or simply $E$ to each work band $W \in \mathcal{W}$. Fix one particular work band $W$. The evaluation band $E_W$ is the set of vertices $w \in W$ for which the $s$-star stencil $S_s(w)$ can be evaluated if all vertices of $W$ would reside in internal memory,

$$E_W = \{ w \in W : S_s(w) \subset W \}.$$

For an infinite work band the infinite evaluation $E^\infty$ band is defined in the analogous way. Similarly to associating an evaluation band $E_W$ with a work band $W$, we associate a work band $W_E$ with an evaluation band $E$. If an algorithm evaluates $s$-star stencil for all grid points, the evaluation bands have to cover the grid.

As an evaluation band is a true subset of its corresponding work band, the work bands have to overlap if the evaluation bands shall cover the grid. Because of this overlap there are going to be vertices that belong to several work bands. These vertices that are part of several different work bands cause the non-compulsory I/Os.

We introduce the notion of $k$-intersections to partition the vertices of the grid according to the work bands and evaluation bands they belong to. The $k$-intersections are fundamental for the data layout and allow a simple counting of the non-compulsory I/Os. To define the $k$-intersections, let $\mathcal{W}$ be the set of all work bands that an algorithm works on. For $k \in \mathbb{N}$ and two subsets $\mathcal{E}'$ and $\mathcal{W}'$ such that $\mathcal{E}' \subset \mathcal{W}' \subset \mathcal{W}$ and $|\mathcal{W}'| = k$, the $k$-intersection $\Phi(\mathcal{W}', \mathcal{E}')$ is the set of all vertices which
belong to all $W'$ for $W' \in W'$ and all $E_W$ for $W \in E'$, but not to any other work or evaluation bands,

$$\Phi(W', E') := \Phi(W') \cap \Phi^E(E')$$

for

$$\Phi(W') := \left( \bigcap_{W \in W'} W \right) \setminus \left( \bigcup_{W \not\in W'} W \right)$$

and

$$\Phi^E(E') := \left( \bigcap_{W \in E'} E_W \right) \setminus \left( \bigcup_{W \not\in E'} E_W \right).$$

We call $\Phi(W')$ work band intersection and $\Phi^E(E')$ evaluation band intersection. For a fixed $k \in \mathbb{N}$ and a particular work band $W \in W$ the family of all $k$-intersections which contain vertices of $W$ is given by

$$\Phi(W, k) := \left\{ \Phi(W', E') : W', E' \subset W, \ E' \subset W', \ |W'| = k \text{ and } W \in W' \right\}.$$  

The $k$-intersections describe sets of vertices that are going to be either read or written in sequence to achieve good performance. To use the compulsory reads effectively, the data is split according to the $\Phi(W')$. To make sure that the compulsory write can store a whole block of vertices to external memory, the data is organized by the $\Phi^E(E')$. To avoid overhead with respect to both, compulsory reads and writes, the data is divided according to the $k$-intersections $\Phi(W', E')$. In the presented algorithms the vertices of the 1-intersections are only going to contribute to the compulsory I/Os and the vertices of the 2-intersections are going to determine the leading term of the non-compulsory I/Os. The I/Os caused by vertices in the $k$-intersections for $k \geq 3$ will only amount to lower order terms.

To describe the size of the sweep shape and the resulting work band a parameter $m$ sufficing

$$|\mathcal{S}| = c \cdot m^{d-1} + O\left(m^{d-2}\right)$$

for a constant $c \in \mathbb{R}$ is employed. For our polygonal sweep shapes a natural choice for $m$ is the width of $\mathcal{S}$ in one unit direction, e.g. $m = \max\{|u_1 - v_1| : u, v \in \mathcal{S}\}$. The size parameter $m$ is going to be specified for each individual sweep shape. The size of a work band is going to be chosen so that we can evaluate the corresponding evaluation band by doing one sweep of the work band, i.e. by loading each vertex of the work band exactly once.

For a sweep shape $\mathcal{S}$ and the resulting work band $W$ and evaluation band $E_W$, all vertices of $\mathcal{S} \cap E_W$ can be evaluated if the $s$ preceding and $s$ proceeding sweep shapes of $\mathcal{S}$ are in internal memory. Hence an
evaluation band can be evaluated by one sweep of the work band when the internal memory can hold $2s + 1$ sweep shapes. When the vertices within a sweep shape are not evaluated randomly but in lexicographic order, only vertices equivalent to $2s$ full sweep shapes ($\mathcal{O}(m^{d-1})$ vertices each) and an overhead of $\mathcal{O}(m^{d-2})$ vertices is needed in internal memory instead of $2s + 1$ full sweep shapes. The vertices of the $s$-th preceding sweep shape can be deleted or written to the external memory as vertices of the $s$-th proceeding sweep shape are loaded. For the leading term of the non-compulsory I/Os only the number sweep shapes is relevant. The overhead of $\mathcal{O}(m^{d-2})$ vertices is only going to contribute to lower order terms.

To prove that the sweep shape and the resulting work band are small enough to evaluate the evaluation band with a single sweep of the work band, the notion of a work band order is introduced. Fix one work band $W$. The vertices $w \in W$ are sorted in two stages:

1. Sweep shape by sweep shape in increasing distance of the sweep shape to the origin.

2. Within sweep shapes in lexicographic order.

The lexicographic order is the lexicographic order with respect to the coordinates of the vertices, $x_d$ being the index changing fastest and $x_1$ the slowest index. Formally,

$$w \leq w' \iff \left( \exists j \in \{1, \ldots, d\} : \forall i < j : w_i = w'_i \land w_j < w'_j \right) \lor \left( w_i = w'_i \forall i \in \{1, \ldots, d\} \right).$$

For a vertex $w \in W$ its work band position or work band order $o_W(w)$ is its position in the work band according to this order. For two vertices $w, w' \in W$ their distance in the work band order is

$$\|w - w'\|_W = |o(w) - o(w')|,$$

i.e. the difference of their respective positions within this work band $W$. Note that vertices which are in $k$-intersections for $k \geq 2$ belong to more than one work band and hence are assigned a work band order for each of the work bands they belong to. Also, vertices which do not belong to the same work band are not comparable.

In the very same way as the work band order $o_W(\cdot)$, the evaluation band order $o_E(\cdot)$ and the $k$-intersection order $o_{\Phi(W', E)}(\cdot)$ are defined for all vertices $w \in E$ or $w \in \Phi(W', E)$ respectively. As a consequence, the orders of different work bands, evaluation bands and $k$-intersections are consistent with each other. Formally, let $A$ and $B$ be either be work
bands, evaluation bands or k-intersections. The orders of A and B are called consistent if and only if:

\[ \forall w, w' \in A \cap B : \ (o_A(w) \leq o_A(w')) \iff (o_B(w) \leq o_B(w')) . \]

The linear work band order gives rise to the definition of an interval of vertices of the work band \( W \). For \( \delta \in \mathbb{N} \),

\[ [-\delta + w, w + \delta]_W := \{ w' \in W : \| w' - w \|_W \leq \delta \} \]

is the interval of midpoint \( w \) and width \( 2\delta \). The vertices needed to evaluate a vertex \( w \in E_W \) of the evaluation band \( E_W \) are contained in the interval

\[ I(w) := \left\{ w' \in W : \exists v \in S_s(w), \exists v' \in S_s(w) \text{ such that:} \right\}
\[ o_W(v) \leq o_W(w') \leq o_W(v') \]. \quad \text{(3.16)} \]

By definition, \( S_s(w) \subset I(w) \). Note that if \( w \in E_W \) then it also holds that \( w \in W \). Furthermore, it follows from \( w \in E_W \) that the whole s-star stencil of \( w \) is a subset of \( W \), i.e. \( S_s(w) \subset W \), and the definition above is well defined. Finally, observe that there is a simple characterization of \( I(w) \) following directly from the definition. Denote by \( w_{\min} \) and \( w_{\max} \) the vertices in \( S_s(w) \subset W \) of the smallest respectively largest work band order, i.e.

\[ w_{\min} = \arg\min \{ o_W(w) : w \in S_s(w) \subset W \} \quad \text{and} \]
\[ w_{\max} = \arg\max \{ o_W(w) : w \in S_s(w) \subset W \} . \quad \text{(3.16)} \]

Then, \( I(w) \) is given by

\[ I(w) = [o_W(w_{\min}), o_W(w_{\max})] \quad \text{(3.17)} \]

To facilitate the analysis, we reduce the dimensionality of the problem to \( d - 1 \) by cutting it with hyperplanes of normal \( e_1 \). In particular, the vertices of the k-intersections causing the non-compulsory I/Os are going to be counted layer by layer, hyperplane of normal \( e_1 \) by hyperplane of normal \( e_1 \). The counting is done in a two-step process. First the number of work bands that are in one hyperplane is bounded. Then, for each hyperplane and each work band the number of vertices in any k-intersection is bounded. For both tasks, we need more notation.

First, let us introduce notation to estimate the number of work bands in a particular hyperplane of normal \( e_1 \). Denote by \( H_h \) the hyperplane of normal \( e_1 \) at distance \( h \) from the origin and let \( \mathbb{W} \) be a list of work bands. The work bands that have a vertex in \( H_h \) are denoted by \( \mathbb{W}_h \),

\[ \mathbb{W}_h = \{ W \in \mathbb{W} : W \cap H_h \neq \emptyset \} \]
Estimating the number of work bands in $\mathcal{W}_h$ is the first part of the complexity analysis. Let $E^\infty$ be an infinite evaluation band. The intersection of the evaluation band at level $h$ is then defined as

$$E^\infty_h = E^\infty \cap H_h.$$  \hfill (3.18)

(In general, for sets $A$ the subscript $h$ is used as a shortcut for $A_h = A \cap H_h$, i.e. for all vertices of $A$ for which $x_1 = h$ holds.) For simple sweep sequences it holds that $E^\infty_{h_1} = E^\infty_{h_2}$, up to translations for all $h_1, h_2 \in \mathbb{Z}$. We say that a set of work bands is created by the same sweep shape (and sweep sequence) when only translations of one particular sweep shape are used to create the different work bands. Given that the work bands are created by the same sweep shape and sweep sequence, all corresponding infinite evaluation bands have the same cross-sections up to translations. In particular, for two infinite evaluation bands $E^\infty$ and $(E')^\infty$ whose corresponding work bands are given by the same sweep shape and sweep sequence, the sizes of their level sets at height $h$ and $h'$ are equal,

$$|(E^\infty)_h| = |((E')^\infty)_{h'}| \quad \forall E, E' \quad \forall h, h' \in \mathbb{Z}.$$  

For a fixed sweep shape determining $m$, a fixed sweep sequence and a list of work bands $\mathcal{W}$ that is created by this sweep shape and sweep sequence, there is a constant $e \in \mathbb{R}$, $e \geq 0$ such that

$$\forall h \in [k_1], \forall W \in \mathcal{W}: |(E_W)^\infty_h| \geq e \cdot m^{d-1} - O(m^{d-2}).$$  \hfill (3.19)

Such an $e$ always exists as $e = 0$ may be chosen. We are going to derive the value of $e$ in the sections of the different algorithms for the respective sweep shapes and sweep sequences.

Finally, denote by $l_i$ ($2 \leq i \leq d$) the width of $E^\infty_h$ in direction $i$, i.e.

$$l_i = \max_{x \in E^\infty_h} \{x_i\} - \min_{x \in E^\infty_h} \{x_i\}.$$  \hfill (3.20)

Again, given that the same sweep shape and sequence was used to create different work bands, $l_i$ is independent of the actual choice of $E$ and the level $h$.

It is left to count the vertices in the $k$-intersections of a work band per hyperplane of normal $e_1$. Let $\mathcal{W}$ be a list of work bands and choose one $W \in \mathcal{W}$. For $k \in \mathbb{N}$ and $h \in [k_1]$ the vertices of the $k$-intersections of $W$ at height $h$ are given by

$$\Phi(W, k, h) = \Phi(W, k) \cap H_h.$$
3.6.2 Algorithmic Framework for the Band Algorithms

With this notation at hand, we are ready to define the algorithmic framework of the band algorithms presented in this chapter. Recall the definitions and assumptions we need for the asymptotic analysis. The dimension \( d \) and the stencil size \( s \) are assumed to be fixed and constant. The grid sizes \( k_1(n) \) are ordered by size, i.e. \( k_1(n) \geq k_2(n) \geq \cdots \geq k_d(n) \), and we assume \( M(n) = o(k_d(n)) \) and \( B(n) = o(M(n)) \) for \( n \to \infty \).

All algorithms work on two copies of the data, an input and an output grid. The input grid stores the initial values of the vertices and these values are never altered. The values updated according to the \( s \)-star stencil are stored in the output grid. When we have to evict blocks of the input data from internal memory due to capacity reasons, we store them back to external memory if that block is accessed again by the algorithm. If the block is not accessed again by the algorithm, it is discarded and not written back to external memory. Output blocks are always stored in external memory. The first time we access an output block it does not need to be read, though, as it does not contain any data.

**Definition 3.11 (Band Algorithm).** A Band Algorithm \( A \) evaluating the \( s \)-star stencil on the \( d \)-dimensional grid \([k_1] \times \cdots \times [k_n] \) is defined by

1. a simple sweep sequence \( \mathcal{X} \),
2. a sweep shape \( \mathcal{I} \) and
3. a list \( \mathcal{W} \) work bands.

All work bands \( W \in \mathcal{W} \) need to be generated by the sweep shape \( \mathcal{I} \) and sweep sequence \( \mathcal{X} \).

The algorithm works on a data layout organized

1. by \( k \)-intersections,
2. within \( k \)-intersection in \( k \)-intersection order (i.e. sweep shape by sweep shape in increasing distance to the origin and within sweep shapes by the lexicographic order of the coordinates).

The algorithm evaluates the vertices in the following order:

1. Evaluation band by evaluation band in the order of the corresponding work bands in the list \( \mathcal{W} \).
2. Within an evaluation band in the evaluation band order (i.e. sweep shape by sweep shape in increasing distance to the origin and within sweep shapes in lexicographic order).

If the \( k \)-intersection of an evaluation band has already been evaluated, it is not evaluated again.

If a vertex \( w \in E \) of a particular evaluation band \( E \) is evaluated, all blocks that store the input values of the vertices of the interval \( I(w) \) are loaded to
internal memory. The least recently used (LRU) cache replacement strategy is used when input blocks have to be evicted from internal memory. Output blocks containing the updated values of a vertex are only evicted from internal memory if the updated values of all vertices of the block have been calculated or the end of the work band has been reached.

**Definition 3.12** (Correct and Memory Efficient Band Algorithm). Let \( A \) be a Band Algorithm as given in Definition 3.11. Let \( m \) be a parameter for the size of the sweep shape that suffices Equation 3.15, i.e.

\[
|\mathcal{S}| = c \cdot m^{d-1} + O\left(m^{d-2}\right)
\]

for a constant \( c \in \mathbb{R}_{>0} \). We then choose the size \( m \) of the sweep shape as

\[
m = \frac{a^{-1}}{2s \cdot c} \cdot \Theta \left(a^{-\sqrt{B}}\right) .
\]

(3.21)

\( A \) is called memory efficient if the following assumptions hold:

1. The interval of a vertex \( w \in W \) is small:
   \[
   \forall W \in \mathbb{W} \text{ and } \forall w \in E_W \text{ it holds that } I(w) \subset [x - \delta, x + \delta]_W \text{ for } \delta = s \cdot |\mathcal{S}| + O \left(M \frac{d-2}{d-1}\right).
   \]

2. The evaluation bands cover the grid:
   \[
   \forall w \in [k_1] \times \cdots \times [k_n] : \exists W \in \mathbb{W} \text{ such that } w \in E_W.
   \]

3. The width of an evaluation band is small:
   \[
   l_i = O(m) \quad \forall i \in \{2, \ldots, d\}.
   \]

4. Size of the evaluation bands: \( \exists e \in \mathbb{R}_{>0} \) such that Equation 3.19 holds, i.e.
   \[
   \forall h \in [k_1], \forall W \in \mathbb{W} : \left|\left(E_W\right)_h^\infty\right| \geq e \cdot m^{d-1} - O \left(m^{d-2}\right).
   \]

5. Work band vertices are not separated from the evaluation band:
   \[
   \forall W \in \mathbb{W} \text{ and } \forall w \in W : \exists v \in E_W \text{ s.t. } w_1 = v_1 \text{ and } \|w - v\|_1 \leq 2s.
   \]

6. The total number of work bands is small:
   \[
   |\mathbb{W}| = O \left(\frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right).
   \]

7. Any work band overlaps only with a constant number of other work bands:
   \[
   \forall W \in \mathbb{W} : \left|\{V \in \mathbb{W} : (V \neq W) \land (V \cap W \neq \emptyset)\}\right| = O(1).
   \]

8. The 2-intersections determine the leading term of the non-compulsory I/Os:
   \( \exists b \in \mathbb{R}_{>0} \) such that \( \forall h \in [k_1], \forall W \in \mathbb{W} : \)
   \[
   \text{For } d \geq 3: \left|\Phi(W, 2, h)\right| \leq b \cdot m^{d-2} + O \left(m^{d-3}\right).
   \]
   \[
   \text{For } d = 2: \left|\Phi(W, 2, h)\right| \leq b.
   \]
9. The k-intersections for $k \geq 3$ only contribute to lower order terms of the non-compulsory I/Os: $\forall h \in [k_1], \forall W \in W$, for $k \geq 3$:

For $d \geq 3$: $|\Phi(W,k,h)| = O(m^{d-3})$.

For $d = 2$: $|\Phi(W,k,h)| = 0$.

**Theorem 3.13** (The I/O Complexity of a Memory Efficient Band Algorithm). Let $A$ be a memory efficient band algorithm as defined by **Definition 3.11** and **Definition 3.12**. Then, an upper bound for the non-compulsory I/Os performed by $A$ is given by

$$d = 2: \frac{b \cdot c}{e} \cdot 2s \cdot \frac{k_1 k_2}{B \cdot M} + O\left(\frac{k_1 k_2}{M^2}\right) + O\left(\frac{k_1}{B}\right),$$

$$d \geq 3: \frac{b \cdot a^{-\sqrt{c}}}{e} \cdot a^{-\sqrt{2s}} \cdot \frac{\prod_{i=1}^{d} k_i}{B \cdot a^{-\sqrt{M}}} + O\left(\frac{\prod_{i=1}^{d} k_i}{\sqrt{B^d M^2}}\right).$$

For any dimension $d$, the number of compulsory I/Os is, by definition, $2 \cdot \frac{1}{B} \cdot \prod_{i=1}^{d} k_i$.

For $d \geq 3$, the leading error term $O\left(\prod_{i=1}^{d} k_i / a^{d-\sqrt{B^d M^2}}\right)$ is due to reserving separate blocks for each k-intersection. For $d = 2$, the additional error term $O\left(\frac{k_1}{B}\right)$ error is due to the estimation of the number of work bands and the last, probably incomplete, work band that the algorithm works on.

We claim that this analysis is tight and that **Theorem 3.13** hence gives the complexity of a memory efficient band algorithm. To prove the theorem, we first prove several Lemmas. Once the theorem is proven, we present different band algorithms in **Section 3.7** and show that these are memory efficient. Hence, their complexity is given by **Theorem 3.13**.

**Lemma 3.14** (The number of non-empty k-intersections of each work band is bounded). Given the setup of **Definition 3.11**, i.e. a sweep shape $\mathcal{S}$, a sweep sequence and a list of work bands $\mathcal{W}$, assume that **Assumption 7** of **Definition 3.12** holds. Then, for any $W \in \mathcal{W}$ the number of k-intersections containing vertices of $W$ is bounded by a constant which only depends on the dimension $d$ and the size of the stencil $s$.

**Proof.** We need to prove that the family

$$\{A : A \in \Phi(W',E') \text{ for } W' \in \mathcal{W}, E' \in \mathcal{W} \text{ and } A \cap W \neq \emptyset\}$$

contains only a constant number of sets. Denote by $O_W$ the set of work bands which overlap with $W$ non-trivially, including $W$ itself. Denote by $\mathcal{P}(O_W)$ the power set of $O_W$. By **Assumption 7** of **Definition 3.12** one work band overlaps non-trivially with only a constant number of other work bands. This constant does only depend on the dimension $d$.
and the size of the stencil \( s \). Hence, \( |O_W| = \Theta(1) \) and therefore it also holds that \( |P(O_W)| = \Theta(1) \). For any non-empty \( k \)-intersection \( \Phi(W', E') \) containing at least one vertex of \( W \) it follows that \( W' \in P(O_W) \). As an evaluation band \( E \) is always a subset of its work band, \( E \subseteq W_E \), it also follows that \( E' \in P(O_W) \). Hence, their is just a constant number of non-empty \( k \)-intersections \( \Phi(W', E') \) that contain vertices of \( W \). \( \square \)

**Lemma 3.15** (The interval \( I(w) \) of a vertex \( w \) and a constant number of output blocks fit into internal memory). *Given the setup of Definition 3.11, i.e. a sweep shape \( \mathcal{S} \), a sweep sequence and a list of work bands \( \mathcal{W} \). Assume that Assumption 1 and Assumption 7 of Definition 3.12 hold. Then, for any work band \( W \in \mathcal{W} \) and any vertex \( w \in W \), the vertices of the interval of \( w \) fit into internal memory together with a constant number of output blocks of size \( B \),

\[
I(w) = M - \Omega(1) \cdot B.
\]

**Proof.** By Assumption 1 of Theorem 3.13 we got the inclusion

\[
I(w) \subset [x - \delta, x + \delta]_W \quad \text{for} \quad \delta = s \cdot |\mathcal{S}| + \mathcal{O} \left( M^{\frac{d-2}{d-1}} \right).
\]

The interval \([x - \delta, x + \delta]_W\) itself contains \( 2\delta + 1 \) vertices. However, the interval is split into several chunks of contiguous memory blocks by the \( k \)-intersection. Denote by \( \Phi(W', E') \) a \( k \)-intersection that contains a vertex of \( I(w) \). Although each \( k \)-intersection is contiguous in memory, all \( k \)-intersections of \( W \) are most likely not contiguous in memory together. Hence, we do not only need to account for the vertices in \([x - \delta, x + \delta]_W\) but also for all vertices which are in the same blocks as any of these vertices. As the work band and the \( k \)-intersection orders are consistent, the vertices of \( I(w) \cap \Phi(W', E') \) are contiguous in memory. Therefore, in at most 2 blocks of \( I(w) \cap \Phi(W', E') \), the first and the last block, are also vertices not from \( I(w) \). Hence, per \( k \)-intersection there are less than 2 blocks in internal memory which contain vertices not in \( I(w) \). By Lemma 3.14, there the number of \( k \)-intersections per work band is constant. Hence, the union of all blocks containing vertices of \( I(w) \) consist of at most \((2 \cdot \delta + 1) + \mathcal{O}(B)\) vertices altogether. Hence proving \( 2 \cdot \delta + 1 + \mathcal{O}(B) = M - \Omega(B) \) establishes the lemma.

\[
2\delta + 1 + \mathcal{O}(B) = 2 \cdot \left( s \cdot |\mathcal{S}| + \mathcal{O} \left( M^{\frac{d-2}{d-1}} \right) \right) + 1 + \mathcal{O}(B) = \\
\overset{(3.15)}{=} 2 \cdot s \cdot \left( c \cdot m^{d-1} + \mathcal{O} \left( m^{d-2} \right) \right) + \mathcal{O} \left( M^{\frac{d-2}{d-1}} \right) + \mathcal{O}(B) = \\
\overset{(3.21)}{=} 2 \cdot s \cdot c \cdot \left( \sqrt[d-1]{\frac{M}{2s \cdot c}} - \Theta \left( \frac{d-1}{\sqrt[d-1]{B}} \right) \right)^{d-1} + \\
+ \mathcal{O} \left( M^{\frac{d-2}{d-1}} \right) + \mathcal{O}(B) =
\]
\[
= M - \Theta \left( M^{\frac{d-1}{2}} \cdot B^{\frac{1}{d-1}} \right) + O \left( M^{\frac{d-1}{2}} \right) + O \left( B \right) = \\
= M - \Theta \left( \frac{d}{d-1} \sqrt{M^{d-2}} \cdot B \right) = M - \Omega \left( B \right).
\]

Lemma 3.16 (Number of work bands per hyperplane of normal $e_1$). Given the setup of Definition 3.11, i.e. a sweep shape $\mathcal{S}$, a sweep sequence and a list of work bands $\mathcal{W}$. Assume that Assumption 3, Assumption 5, Assumption 7, Assumption 8 and Assumption 9 of Definition 3.12 hold. Then, for all $h \in \lfloor k_1 \rfloor$, the hyperplane of distance $h$ from the origin and normal $e_1$ contains vertices of at most

\[
\prod_{i=2}^{d} (k_i + O(m))
\]

\[
\left| E_{\infty}^h \right| + O(m^{d-2})
\]

different work bands.

Proof. Recall the definition of $E_{\infty}^h$ and $l_i$, namely Equation 3.18 and Equation 3.20. In particular note that the values of neither of them depends on the choice of the work band $W_E$ or the level $h$. Let $K$ be a level set of the grid $\lfloor k_1 \rfloor \times \cdots \times \lfloor k_n \rfloor$ in $e_1$ direction,

\[
K = \{h\} \times \lfloor k_2 \rfloor \times \cdots \times \lfloor k_d \rfloor.
\]

Hence, $|K| = \prod_{i=2}^{d} k_i$.

To estimate the number of work bands in a hyperplane of normal $e_1$ we make a detour to the evaluation bands. We first try to estimate the number of evaluation bands needed to cover $K$ by $|K|/|E_{\infty}^h|$: dividing $|K|$ by the number of vertices that can be evaluated per work band and hyperplane of normal $e_1$, namely $|E_{\infty}^h|$. The situation is depicted for the Diagonal Band Algorithm of Section 3.7.2.1 in Figure 3.1. The fraction $|K|/|E_{\infty}^h|$ would underestimate the number of work bands that have a vertex in $K$ for three reasons. First, if the simple sweep sequence also contains a shift different from $e_1$, it is possible that a hyperplane of normal $e_1$ contains a vertex of the work band but none of the corresponding evaluation band. Second, the straight boundary of the grid cannot be reproduced if $E_{\infty}^h$ does not have the same straight boundary, i.e. whenever the sweep shape is more complicated. Third, the irregular structure of $E_{\infty}^h$ could make it impossible to align different evaluation bands without overlap. $|K|/|E_{\infty}^h|$ would assume that perfect partitioning is possible and would allow fractions of evaluation bands to cover $K$. However, we are interested in the number of evaluation needed and are not allowed to add fractional evaluation bands together to form a single one.

We therefore enlarge the grid in two steps to get an upper bound for the number of work bands involved in each hyperplane of normal $e_1$. To address the first issue, pad 2s grid points at the beginning and end of
each coordinate direction $x_i$ for $2 \leq i \leq d$ of the grid. By Assumption 5, this ensures that for every vertex of the work band in the original grid there is also a vertex of the evaluation band in the same hyperplane of normal $e_1$ in the extended grid. Hence, by counting the evaluation bands in the extended grid for a hyperplane of normal $e_1$, we get a lower bound for the work bands in the original grid in this hyperplane.

To address the second issue, pad another $l_i$ grid points at the beginning and end of unit direction $x_i$ for all $2 \leq i \leq d$. Hence, if there is a vertex of an evaluation band $E$ in the hyperplane of normal $e_1$ before this second enlargement, then all of $E$ is in the twice enlarged grid. Therefore, each fractional evaluation band from before is now completely in the grid.

Lastly, to address the third issue, we do not divide $|K|$ by $|E^\infty_h|$ but by the number of vertices in $|E^\infty_h|$ that belong to the 1-intersection and hence are not part of any other evaluation band. By Assumption 8 at most $b \cdot m^{d-2} + \Omega(m^{d-3})$ vertices of $|E^\infty_h|$ belong to 2-intersections (at most $b$ for $d = 2$). By Assumption 7 at most a constant number of work bands, say $f$ work bands, overlap and hence only $k$-intersections for $k$ up to $f$ can be non-empty. Using Assumption 9, the number of vertices of $|E^\infty_h|$ which are in $k$-intersections for $k \geq 2$, the 1-intersection of $|E^\infty_h|$ still contains at least

\[
\begin{align*}
d = 2: & \quad |E^\infty_h| - b = |E^\infty_h| - \Omega(m^{d-2}) \\
d \geq 3: & \quad |E^\infty_h| - \left(\left(b \cdot m^{d-2} + \Omega(m^{d-3})\right) + \Omega(m^{d-3})\right) = \\
& = |E^\infty_h| - \Omega(m^{d-2})
\end{align*}
\]

vertices.

With these three modifications we ensure that the number of work bands containing a vertex of $K$ is bounded above by

\[
\prod_{i=2}^{d} \frac{(k_i + 2 \cdot l_i + 4s)}{|E^\infty_h| - \Omega(m^{d-2})} \quad \text{(Assumption 3)} \quad \prod_{i=2}^{d} \frac{(k_i + \Omega(m))}{|E^\infty_h| - \Omega(m^{d-2})}.
\]
3.6 Notation and Algorithmic Framework for the Upper Bounds

We are now ready to proof Theorem 3.13.

**Proof of Theorem 3.13.** The proof proceeds in the following steps: first, the correctness of a memory efficient band algorithm is proven, making sure all vertices are evaluated and that the internal memory stores the relevant information to evaluate the current vertex. Then, we analyze the I/O complexity of a memory efficient band algorithm. We first determine that a memory efficient band algorithm evaluates an evaluation band by sweeping through the \( k \)-intersections of the corresponding work band simultaneously, loading each vertex of the work band exactly once. From that it follows that the \( k \)-intersections determine the number of non-compulsory I/Os that a vertex causes. Hence, we estimate the total number of vertices in the \( k \)-intersections for different \( k \). After accounting for incomplete blocks at the beginning and end of each \( k \)-intersection, we use these results to establish the upper bound.

The correctness of algorithm \( \mathcal{A} \) is verified easily. For any \( W \in \mathcal{W} \) choose any \( w \in E_W \). First, the interval \( I(w) \) contains all vertices necessary to compute the \( s \)-star stencil of \( w \) as \( S_s(w) \subset I(w) \) by the definition of \( I(w) \). Second, Lemma 3.15 shows that the internal memory is large enough to hold all blocks of \( I(w) \) and a constant number of output blocks. By Lemma 3.14, the number of \( k \)-intersections of \( W \) is constant and hence, as we need at most one output block per \( k \)-intersection, also the number of output blocks. Hence, all blocks containing vertices of \( I(w) \) and the output blocks fit in internal memory together. Finally, by Assumption 2 the evaluation bands cover the grid. As \( \mathcal{A} \) works through all evaluation bands it also evaluates all vertices of the grid and hence performs one update of the grid according to the \( s \)-star stencil.

Let us now analyze the number of I/Os the algorithm performs.

First, let us establish that \( \mathcal{A} \) evaluates an evaluation band \( E \) loading each vertex of \( W_E \) exactly once. As whole blocks of size \( B \) are always loaded, the first and last block of each \( k \)-intersection of \( W_E \) can also contain vertices not in \( W \). Hence, evaluating \( E \) can also load vertices \( w \notin W \). We disregard these vertices \( w \notin W \) for the moment and account for them separately in Equation 3.27. The algorithm \( \mathcal{A} \) evaluates the vertices of \( E \) in the evaluation band order. To evaluate a vertex \( w \in E \) the algorithm \( \mathcal{A} \) loads the interval \( I(w) \subset W \) into internal memory. We have already seen in the correctness proof of \( \mathcal{A} \) that the internal memory is large enough to hold \( I(w) \) and the constant number of output blocks needed. As the orders of the work band, the evaluation band and the \( k \)-intersections are consistent with each other, this means that \( \mathcal{A} \) sweeps over the \( k \)-intersections of \( W \) simultaneously. Therefore evaluating the
next vertex in $E$ results in loading subsequent vertices of the work band order and evicting prior vertices. Formally, the following holds.

For $w, w' \in E$ with $I(w) = [a, b]$ and $I(w') = [a', b']$

it holds that: $o_E(w) \leq o_E(w') \Rightarrow (a \leq a' \wedge b \leq b')$

Using the cache replacement strategy described in Definition 3.11 ensures that neither input nor output blocks are evicted from memory when they are still needed for the evaluation of $E$. Altogether, although these sweeps happen simultaneously, each vertex of $W$ needs to be read exactly once to evaluate all vertices of $E$. We say, that one one sweep of the work band $W$ suffices to evaluate all vertices of $E$.

The $k$-intersections determine how often a vertex needs to be read. Recall that a vertex of a $k$-intersection is part of exactly $k$ work bands. As the vertex needs to be accessed by these $k$ work bands, it causes the first, compulsory read operation followed by a sequence of $k - 1$ non compulsory writes and reads. As well, the updated value of the vertex is stored by the compulsory write. Altogether, a vertex in a $k$-intersection takes part in $2$ compulsory and $2(k - 1)$ non-compulsory I/Os. This are less than $2$ non-compulsory I/Os for each work band the vertex is part of. It is also possible that a vertex is still in internal memory from the previous work band when it is accessed by the next work band. This, however, would only decrease the number of I/Os performed and is hence disregarded in the analysis.

By now we know that vertices in the 1-intersections cause $0$ non-compulsory I/Os, vertices in the 2-intersections cause $1$ non-compulsory I/O per work band they belong to and vertices in $k$-intersections for $k \geq 3$ cause less than $2$ non-compulsory I/O per work band they belong to. Hence, it is left to estimate the number of vertices in the $k$-intersections for $k \geq 2$. We count a vertex that belongs to $k$ different work bands $k$ times as the vertex causes $1$ non-compulsory I/O per work band if $k = 2$ and less than $2$ non-compulsory I/O per work band if $k \geq 3$. Recall that the number of work bands in $H_h$ is bounded by Lemma 3.16 as

$$\prod_{i=2}^{d} (k_i + \mathcal{O}(m)) \cdot \frac{|E_h^\infty| - \mathcal{O}(m^{d-2})}{|E_h^\infty| - \mathcal{O}(m^{d-2})}.$$ 

First, let us bound the number of vertices in all 2-intersections. For $d \geq 3$ Assumption 8 of Definition 3.12 yields that a hyperplane $H_h$ contains at most

$$\left(b \cdot m^{d-2} + \mathcal{O}(m^{d-3})\right) \cdot \prod_{i=2}^{d} (k_i + \mathcal{O}(m)) \cdot \frac{|E_h^\infty| - \mathcal{O}(m^{d-2})}{|E_h^\infty| - \mathcal{O}(m^{d-2})}.$$
vertices in 2-intersections, counting them once for each work band they belong to. As this holds for all \( h \in [k_1] \), the grid \([k_1] \times \cdots \times [k_d]\) contains at most
\[
\kappa_1 \cdot \left( (b \cdot m^{d-2} + \Theta (m^{d-3})) \cdot \prod_{i=2}^{d} \left( k_i + \Theta (m) \right) \right) \left( E_h^\infty - \Theta (m^{d-2}) \right) \tag{3.24}
\]
vertices in 2-intersections for \( d \geq 3 \), counting them once for each work band they belong to. For \( d = 2 \) the same argument yields that there are at most
\[
\kappa_1 \cdot \left( (b \cdot m^{d-2}) \cdot \prod_{i=2}^{d} \left( k_i + \Theta (m) \right) \right) \left( E_h^\infty - \Theta (m^{d-2}) \right) \tag{3.25}
\]
vertices in 2-intersections counting them once for each work band they belong to.

Similarly, the vertices in the \( k \)-intersections for \( k \geq 3 \) can be bounded. Fix a \( k \geq 3 \). For \( d = 2 \) these \( k \)-intersections are empty by Assumption 9 of Definition 3.12. So consider \( d \geq 3 \). Similarly to the 2-intersections, Assumption 9 of Definition 3.12 yields that the grid \([k_1] \times \cdots \times [k_d]\) contains at most
\[
\kappa_1 \cdot \Theta (m^{d-3}) \cdot \prod_{i=2}^{d} \left( k_i + \Theta (m) \right) \left( E_h^\infty - \Theta (m^{d-2}) \right) \tag{3.26}
\]
vertices in the \( k \)-intersection for a fixed \( k \geq 3 \), counting each vertex once for each work band it belongs to.

Now, consider the I/Os necessary to load the first and last block of each \( k \)-intersection of \( W_E \) in general containing vertices \( w \notin W \). We disregarded loading these \( w \notin W \) vertices up to now when we said that \( E \) can be evaluated by one sweep of the work band \( W_E \). All in all, these are are less than 2 blocks or \( 2 \cdot B \) vertices for each \( k \)-intersection of a work band. For each block we account for 2 non-compulsory I/Os, one read and one write operation. The number of work bands is given by Assumption 6 as \(|W| = \Theta \left( \frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right) \). By Lemma 3.14 the number of \( k \)-intersections per work band is constant. Hence, the total number of non-compulsory I/Os caused by incomplete blocks at the beginning and end of the \( k \)-intersections is bounded by
\[
|W| \cdot 2 \cdot 2 \cdot \Theta (1) = \Theta \left( \frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right) \tag{3.27}
\]
We are now ready to establish the upper bound for the non-compulsory I/Os of the memory efficient band algorithm \( A \). Accounting for the extra non-compulsory I/Os for incomplete blocks at the beginning and end of each \( k \)-intersection with Equation 3.27, we can assume that we
evaluate $E$ by sweeping through all $k$-intersections of $W_E$ simultaneously. As we access the vertices of the $k$-intersections in their $k$-intersection order, we can always make use of the full block of data loaded. Recall that vertices of $2$-intersections take part in 1 non-compulsory I/O for their work band and the vertices of $k$-intersections for $k \geq 3$ take part in less than 2 non-compulsory I/Os for each of their work bands. Also, by Assumption 7 only $k$-intersections for $k$ up to some constant $f$ can be non-empty.

If a vertex belongs to several work bands, it is already counted multiple times in Equation 3.24, Equation 3.25 and Equation 3.26. Hence, an upper bound for the non-compulsory I/Os of algorithm $A$ for $d \geq 3$ is given by

\[
1 \cdot \frac{1}{B} \cdot k_1 \cdot \left( b \cdot m^{d-2} + \Theta(m^{d-3}) \right) \cdot \frac{\prod_{i=2}^{d} (k_i + \Theta(m))}{|E_h| - \Theta(m^{d-2})} + \\
+ \sum_{j=3}^{f} \left( 2 \cdot \frac{1}{B} \cdot k_1 \cdot \left( m^{d-3} \right) \cdot \frac{\prod_{i=2}^{d} (k_i + \Theta(m))}{|E_h| - \Theta(m^{d-2})} \right) + \\
+ \Theta \left( \frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right) \quad \text{(Assumption 4)}
\]

\[
\frac{k_1}{B} \cdot \frac{\prod_{i=2}^{d} k_i}{e \cdot m^{d-1} - \Theta(m^{d-2})} \cdot \left( b \cdot m^{d-2} + \Theta(m^{d-3}) \right) + \\
+ \Theta \left( \frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right) \quad \text{(Assumption 3)}
\]
\[ b \cdot \prod_{i=1}^{d} k_i = \frac{1}{e \cdot m - \Theta(1)} + O \left( \prod_{i=1}^{d-1} k_i \right) + O \left( \frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right) = \]

\[ \prod_{i=1}^{d} k_i \cdot \frac{1}{e \cdot B} \cdot \frac{1}{d-1} \sqrt{\frac{M}{2s^2c}} - \Theta \left( \frac{d-1}{\sqrt{B}} \right) + O \left( \prod_{i=1}^{d-1} k_i \right) + O \left( \frac{1}{B} \cdot \prod_{i=1}^{d-1} k_i \right) = \]

\[ = b \cdot \prod_{i=1}^{d} k_i \cdot \frac{1}{e \cdot \frac{d-1}{\sqrt{2s^2c}}} + O \left( \frac{d-1}{\sqrt{B}} \right) + O \left( \frac{1}{B} \cdot \prod_{i=1}^{d-1} k_i \right) + O \left( \frac{1}{M} \cdot \prod_{i=1}^{d-1} k_i \right) = \]

\[ = \frac{b}{e} \cdot \frac{d-1}{\sqrt{2s^2c}} \cdot \frac{d-1}{\sqrt{2s^2c}} \cdot \prod_{i=1}^{d} k_i + O \left( \frac{1}{B} \cdot \prod_{i=1}^{d-1} k_i \right) + O \left( \frac{1}{B} \cdot \prod_{i=1}^{d-1} k_i \right) + O \left( \frac{1}{B} \cdot \prod_{i=1}^{d-1} k_i \right) = \]

For \( d = 2 \), the bound can be deduced by very similar calculations as (just the \( \Theta \left( m^{d-3} \right) \) terms are missing and the last step combining lower order terms is not possible)

\[ \frac{1}{B} \cdot k_1 \cdot b \cdot k_2 + O \left( m \right) + O \left( \frac{k_1}{M} \right) = \]

\[ = \frac{b}{e} \cdot 2s \cdot k_1 k_2 + O \left( \frac{k_1 k_2}{M^2} \right) + O \left( \frac{k_1}{B} \right) \cdot \]

3.7 Upper Bounds

This section gives several memory efficient band algorithms for 2 dimensions, 3 dimensions and arbitrary dimensions. The complexity of these algorithms is given by Theorem 3.13. The leading term of the non-compulsory I/Os of the presented memory efficient band algorithms is asymptotically optimal. The constant of the leading term of the non-compulsory I/Os depends on the choice of the sweep shape and sweep sequence. For 2 dimensions the constant of the leading term of the non-compulsory I/Os of the Diagonal Band Algorithm matches the lower bound. In 3 dimensions, the best constant is by a factor of \( \sqrt{2} \) worse than
<table>
<thead>
<tr>
<th>dD Algorithms</th>
<th>Sweep Shape</th>
<th>Sweep Seq.</th>
<th>Non-Compulsory I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Algorithm</td>
<td>Hypercube</td>
<td>$e_1$</td>
<td>$\Omega \left( \frac{1}{d^{d-1} \sqrt{Bd-2} \cdot M} \prod_{i=1}^{d} k_i \right)$</td>
</tr>
<tr>
<td>Column Algorithm</td>
<td>Hypercube</td>
<td>$e_1$</td>
<td>$\Omega \left( \frac{1}{d \sqrt{M}} \prod_{i=1}^{d} k_i \right)$</td>
</tr>
<tr>
<td>Hypercube Band Alg.</td>
<td>Hypercube</td>
<td>$e_1$</td>
<td>$\frac{4s \cdot d^{d-1} \sqrt{2d \cdot (d-1)}}{B \cdot d^{d-1} \sqrt{M}} \prod_{i=1}^{d} k_i$</td>
</tr>
</tbody>
</table>

| Lower Bound           | n.a.        | n.a.       | $\frac{1}{d^{d-1} \sqrt{d!}} \cdot \frac{4s \cdot d^{d-1} \sqrt{2d \cdot (d-1)}}{B \cdot d^{d-1} \sqrt{M}} \prod_{i=1}^{d} k_i$ |

Table 3.2: Leading term of the non-compulsory I/Os for different algorithms in arbitrary dimension.

the lower bound and in arbitrary dimensions $d$ the best constant differs by a factor of $\frac{d-\sqrt{d!}}{d} \approx \frac{d}{e}$. In addition, we present two standard algorithms, that work on the standard row- and column-major data layout, which are also blocked and show that their performance is asymptotically worse.

3.7.1 General Upper Bounds for Arbitrary Dimensions

This section discusses algorithms that work for arbitrary dimensions. First, we discuss the Hypercube Band Algorithm which takes advantage of the non-standard data layout specified in Theorem 3.13. Then, we analyze two standard algorithms, the row and the Column Algorithm. These algorithms are standard in the sense that they work on the standard row- or column-major layout of the data, respectively. They both work with basic blocks given by the work bands just as the band algorithms do. Due to the standard data layout, however, the Row and the Column Algorithm cannot take advantage of the blocks with respect to the fastest changing dimension. As the Row and the Column Algorithms do not work on the band layout, we cannot take advantage of Theorem 3.13 to analyze their complexity. The leading terms of the non-compulsory I/Os for the Row and Column Algorithms as well as for the Hypercube Band Algorithm are summarized in Table 3.2. The Row Algorithm is factor of $\Theta \left( \frac{d-\sqrt{B}}{B} \right)$ worse then the lower bound and the Column Algorithm by a factor of $\Theta (B)$. The Hypercube Band Algorithm achieves the correct asymptotic of the leading term of the non-compulsory I/Os but is by a constant factor of $\frac{d-\sqrt{d!}}{B} \approx \frac{d}{e}$ worse than the lower bound. As all three algorithms work in particular in the 2- and 3-dimensional setting, they are also fundamental algorithms for the low-dimensional cases.
3.7 Upper Bounds

3.7.1.1 The Hypercube Band Algorithm: Asymptotically Optimal for Arbitrary Dimensions

This section discusses the Hypercube Band Algorithm which is a simple, asymptotically matching upper bound for arbitrary dimension $d$. As the Hypercube Band Algorithm works on a data layout supporting its particular access, its complexity is matching the lower bound asymptotically. The Hypercube Band Algorithm is within the framework of memory efficient band algorithms and hence we can apply Theorem 3.13. For two dimensions, this algorithm is depicted in Figure 3.4.

As sweep sequence $\mathcal{X}$ only the single unit shift $e_1$ is employed. The sweep shape $\mathcal{S}$ is a $d-1$ dimensional hypercube lying in a hyperplane of normal $e_1$ with $m$ vertices in each of the remaining $d-1$ directions,

$$\mathcal{S} = \left\{ x = (x_1, \ldots, x_d) \in [k_1] \times \cdots \times [k_d] : \right. \\
\left. x_1 = 1 \land (1 \leq x_i \leq m \text{ for } 2 \leq i \leq d) \right\}$$

the sweep shape consists of $|\mathcal{S}| = m^{d-1}$ vertices in total. The constant $c$ describing the relation between the size parameter $m$ and the number of vertices in $\mathcal{S}$ in Equation 3.15 is therefore $c = 1$. The intersection of a resulting infinite evaluation band with a hyperplane of normal $e_1$ is a hypercube of side length $m - 2s$. Therefore

$$|E^\infty_{h}| = (m - 2s)^{d-1} = m^{d-1} + 0 \left( m^{d-2} \right)$$

and hence $e = 1$ (Assumption 4). The evaluation bands are characterized by $E^\infty_{h}$ being a $d-1$ dimensional hypercube of side length $m - 2s$ lying in the center $W^\infty_{h}$, i.e. $s$ units away from every side of the work band. Hence Assumption 3 and Assumption 5 of Definition 3.12 is fulfilled.

As the sweep sequence consists only of the unit shift $e_1$, covering the grid $[k_2] \times \cdots \times [k_n]$ with sets $E^\infty_{h}$ yields a cover for the $d$ dimensional grid once the sweep sequence is applied. As $E^\infty_{h}$ is also a hypercube, it is easy to cover the $d-1$ dimensional grid. The grid $[k_2] \times \cdots \times [k_n]$ is covered by the evaluation bands if an evaluation band is placed every $m - 2s$ vertices for the last $d-1$ unit directions in a lattice-like structure.
as shown in Figure 3.3. This specifies the list of work bands $W$. The number of work bands per $H_h$ equals the total number of work bands and is

$$|W| = \prod_{i=2}^{d} \left\lfloor \frac{k_i}{m-2s} \right\rfloor = \Theta \left( \prod_{i=2}^{d} \frac{k_i}{m} \right) = O \left( \frac{1}{M} \cdot \prod_{i=2}^{d} k_i \right).$$

Hence, Assumption 6 is satisfied. By construction, the evaluation bands cover the grid (Assumption 2). One work band $W \in W$ overlaps with at most $3^{d-1}-1 = \Theta(1)$ other work bands and hence Assumption 7 is met.

Lemma 3.17 ensures that Assumption 1 is met.

**Lemma 3.17.** Given the $d$ dimensional grid $[k_1] \times \cdots \times [k_d]$ and the setup of Definition 3.12 assuming that the sweep sequence is $X = \{e_1\}$. It then holds for any $W \in W$ and any $w \in E_W$ that

$$I(w) = [x - \delta, x + \delta] \quad \text{with} \quad \delta = s \cdot |\mathcal{S}|.$$

**Proof.** By Equation 3.17 we know that $I(w) = [o_W(w_{\min}), o(w_{\max})]$ with $w_{\min}$ and $w_{\max}$ as defined in Equation 3.16. Denote by $\mathcal{S}$ the sweep shape of $W$ that contains $w$ and by $\mathcal{S}_k$ the sweep shape that is preceding ($k < 0$) or proceeding ($k > 0$) $\mathcal{S}$ in the work band by $k$ shifts. As we are only sweeping in $x_1$-direction, the only vertex of $S_s(w)$ which is in $\mathcal{S}_{-s}$ is $w - s \cdot e_1$. Also, $S_s(w) \cap \mathcal{S}_k = \emptyset$ for $k < -s$. Hence,

$$w_{\min} = w - s \cdot e_1.$$

As $w_{\min}$ and $w$ have the same lexicographic position within their respective sweep shape we get

$$\|w_{\min} - w\|_W = s \cdot |\mathcal{S}|.$$
Similarly \( w_{\text{max}} = w + s \cdot e_1 \) and \( \|w_{\text{max}} - w\|_W = s \cdot \mathcal{S} \) and hence the claim follows. \( \square \)

It is left to estimate the size of the \( k \)-intersections per hyperplane of normal \( e_1 \), \( \Phi(W; k, h) \) (see Figure 3.3). For one work band, there are \( 2(d-1) \) different 2-intersections, one for each of the faces of the \( d-1 \) dimensional hypercube sweep shape. Per hyperplane of normal \( e_1 \), any 2-intersection contains \( 2s \cdot (m - 4s)^{d-2} \) vertices. In total these are

\[
2(d-1) \cdot 2s \cdot (m - 4s)^{d-2} = 4s(d-1)m^{d-2} + \mathcal{O}(m^{d-3})
\]

vertices if \( d \geq 3 \). For \( d = 2 \) these are

\[
2 \cdot (2-1) \cdot 2s \cdot (m - 4s)^{2-2} = 4s
\]

vertices. Hence, \( b = 4s \cdot (d - 1) \) in both cases and Assumption 8 holds.

If three or more work bands intersect, at least one of them has to be offset from \( W \) in two different unit directions. The intersection of two work bands contains \( 2s \) vertices in direction \( i \) if the work bands are at a offset in this direction and \( m - 4s \) vertices if they are not at offset in this direction. Hence, the number of vertices in any \( k \)-intersection for \( k \geq 3 \) is \( \mathcal{O}(k_1 \cdot m^{d-3}) \) or \( \mathcal{O}(m^{d-3}) \) vertices per hyperplane of normal \( e_1 \). As there are only a constant number of non-empty \( k \)-intersections per work band by Lemma 3.14, also \( \left| \Phi(W; k, h) \right| = \mathcal{O}(m^{d-3}) \) for any \( k \geq 3 \). If \( d = 2 \), only up to two work bands overlap and the \( k \)-intersections for \( k \geq 3 \) are empty. This yields that Assumption 9 is also satisfied.

Hence, Theorem 3.13 can be applied to the Hypercube Band Algorithm with \( c = e = 1 \) and \( b = 4s \cdot (d - 1) \). Therefore, the number of non-compulsory I/Os of the Hypercube Band Algorithm is upper bounded by

\[
4s \cdot \sqrt{2s} \cdot (d - 1) \cdot \frac{\prod_{i=1}^{d} k_i}{B \cdot \sqrt{M}} + \mathcal{O} \left( \frac{\prod_{i=1}^{d} k_i}{\sqrt{B} \cdot M^{d-2}} \right).
\]

For \( d = 2 \) and \( d = 3 \) this bound reads

\[
d = 2: \quad 8s^2 \cdot \frac{k_1k_2}{B \cdot M} + \mathcal{O} \left( \frac{k_1k_2}{M^2} \right) + \mathcal{O} \left( \frac{k_1}{B} \right) \quad \text{and}
\]

\[
d \geq 3: \quad 8 \cdot \sqrt{2} \cdot s^{3/2} \cdot \frac{k_1k_2k_3}{B \cdot \sqrt{M}} + \mathcal{O} \left( \frac{k_1k_2k_3}{\sqrt{B} \cdot M^2} \right).
\]

### 3.7.1.2 Standard Row and Column Algorithms: Baselines With Non-Optimal Asymptotic Behavior

This section analyses two standard approaches to evaluate stencils in arbitrary dimension called Row Algorithm and Column Algorithm. Both
algorithms are depicted in Figure 3.2 for two dimensions. Both algorithms cause an asymptotically non-optimal number of non-compulsory I/Os. Hence, the analysis focuses on the asymptotic number of non-compulsory I/Os only and does neither derive the constant of the leading term nor lower order terms.

The algorithms are standard in the sense that they work on the common data layouts used to stored multidimensional arrays, i.e. grids, and not the layout specified in Theorem 3.13.

These standard layouts are the row- and the column-major layout. To be specific, the first coordinate $x_1$ is changing fastest in the row-major layout and the second coordinate $x_2$ is changing fastest in the column-major layout.

The Row Algorithm and the Column Algorithm are going to sweep a $d - 1$ dimensional hypercube through the grid in $x_1$-direction. Sweeping the hypercube in $x_1$-direction ($x_2$-direction) in a row layout is the same as sweeping the same hypercube in $x_2$-direction ($x_1$-direction) in a column layout. Hence only sweeps in $x_1$-direction will be discussed but both, row and column, layouts. Also, layouts in which the i.th coordinate is changing fastest could be discussed. It only matters, however, whether we sweep along the fastest changing index or not. Hence discussing row and column layouts covers all cases.

The Row Algorithm and the Column Algorithm are specified by their sweep sequence, sweep shape, list of work bands and the data layout they work on. Apart the data layout, the algorithms work precisely as described in Theorem 3.13. However, as the Row and the Column Algorithm use a different data layout than specified in Theorem 3.13 we cannot use this theorem to derive the number of non-compulsory I/Os for these algorithms. The structure and analysis of the algorithms is, however, very similar to the one carried out in Theorem 3.13.
The Row Algorithm is based on a standard row-major layout of the vertices of the grid. For two dimensions, the Row Algorithm is depicted in Figure 3.2. The sweep sequence is $X = \{e_1\}$ and the sweep shape a $d - 1$ dimensional hypercube of side length $m$ lying in a hyperplane of normal $e_1$. As the sweep shape and sweep sequence are the same as for the Hypercube Band Algorithm, the work bands, evaluation bands and list of work bands $W$ are identical to those of the Hypercube Band Algorithm. The sweep shape consists of $m^{d-1}$ vertices. As we sweep solely in $x_1$-direction, the work and evaluation bands extend in this direction. The evaluation bands are characterized by $E_h^\infty$ being a $d - 1$ dimensional hypercube of side length $m - 2s$ lying in the center $W_h^\infty$. Hence the grid is covered by the evaluation bands if an evaluation band is placed every $m - 2s$ vertices for the last $d - 1$ unit directions as shown in Figure 3.3. This specifies the list of work bands $W$. The number of work bands per $H_h$ equals the total number of work bands and is

$$|W| = \prod_{i=2}^{d} \left\lfloor \frac{k_i}{m - 2s} \right\rfloor = \Theta \left( \prod_{i=2}^{d} \frac{k_i}{m} \right).$$

The evaluation order of the vertices as well as the data that is kept in internal memory is as specified in Theorem 3.13.

Let us determine the maximum size $m$ of the sweep shape. Consider the input first. As the blocks of the data layout extend in $x_1$-direction we need to keep at least one block of data in internal memory for each of the $m^{d-1}$ rows of the sweep shape. We assume $B \geq 2s + 1$ and hence one block of data covers $2s + 1$ sweep shapes allowing the middle sweep shape to be evaluated.\(^4\) For the output we need one block of data for each row of the evaluation band, i.e. $(m - 2s)^{d-1}$ blocks. Hence, at least

$$B \cdot (m^{d-1} + (m - 2s)^{d-1}) = \Theta (B \cdot m^{d-1})$$

vertices have to be in internal memory at once. This means that the sweep shape size has to be chosen in the order of

$$m = \Theta \left( \frac{d^{d-1} \sqrt{M}}{B} \right).$$

To determine the number of non-compulsory I/Os consider the $k$-intersections. The data layout is not organized by $k$-intersections but the $k$-intersections still describe the number of non-compulsory I/Os a vertex (or block of vertices) has to cause. A work band exceeds its evaluation band by $2s$ vertices ($s$ vertices before and $s$ vertices after the evaluation

\(^4\) The worst case would be a block aligned data layout, i.e. all blocks start at the same indices of $x_1$ for all rows. In a block aligned data layout it would be necessary to keep the $2s$ previous sweep shapes in internal memory in addition to the $m^{d-1}$ new blocks that are loaded when the end of a set of blocks is reached.
band) in each of the \(d-1\) coordinates \(x_i\) for \(2 \leq i \leq d\). As the evaluation bands cover the grid without overlap, this means that the 1-intersection of each work band (besides the first and last work band in each direction) is characterized by \(\Phi_{(W,1,h_1)}\) being a \(d-1\) dimensional hypercube of side length \((m - 4s)\) lying in the middle of \(W\). Hence per \(H_{h_1}\), the vertices of a work band shared with other work bands is given by

\[
\bigcup_{k=2}^{\infty} \Phi_{(W,k,h_1)} = m^{d-1} - (m - 4s)^{d-1} = \Theta \left( m^{d-2} \right).
\]

With the conservative assumption that all these vertices are only shared by two work bands, each of them causes one non-compulsory I/O for each of the work bands (see also the similar discussion in the proof of Theorem 3.13). Each row of the grid is contiguous in memory and so we have one I/O every \(B\) vertices of each row, or \(k_1/B\) I/Os per row. For the asymptotic analysis we ignore blocks that contain two different rows as this effects only lower order terms.

Hence, the Row Algorithm causes at least

\[
\Theta \left( \prod_{i=2}^{d} \frac{k_i}{m} \right) \cdot \frac{k_1}{B} \cdot \Theta \left( m^{d-2} \right) = \Theta \left( \frac{\prod_{i=1}^{d} k_i}{B} \cdot \frac{1}{m} \right) = \Omega \left( \frac{\prod_{i=1}^{d} k_i}{\sqrt{B} \cdot M} \right)
\]

non-compulsory I/Os. Although we do not provide a proof, we claim that this analysis is tight, i.e. the Row Algorithm causes that many non-compulsory I/Os and the effects disregard only concern lower order terms. For \(d = 2\) and \(d = 3\) this bound reads

\[
\begin{align*}
\text{d = 2: } & \quad \Omega \left( \frac{k_1 k_2}{M} \right) \quad \text{and} \quad \text{d = 3: } \quad \Omega \left( \frac{k_1 k_2 k_3}{\sqrt{B} \cdot M} \right).
\end{align*}
\]

The Row Algorithm causes an asymptotically non-optimal number of non-compulsory I/Os as the row-major data layout forces the algorithm to keep a whole block of \(B\) vertices in internal memory for each row of the sweep shape. To evaluate the current sweep shape, however, at most the \(s\) pre- and \(s\) proceeding sweep shapes are required in memory and hence only a constant number of vertices (at most \(2s + 1\)) per row. This forces the Row Algorithm to choose a relatively small sweep shape size \(m\) and hence decreases the interior-to-boundary ratio of the work bands.

**Column Algorithm.** The Column Algorithms is based on a standard column-major layout of the vertices of the grid. For two dimensions, the Column Algorithm is depicted in Figure 3.2. This means that blocks extend in \(x_2\) direction. The sweep sequence is \(\bar{\mathbf{x}} = \{e_1\}\) and the sweep shape a \(d-1\) dimensional hypercube of side length \(m\) con-
sisting of \( m^{d-1} \) vertices. As the sweep sequence and sweep shape are identical to the Row Algorithm and the Hypercube Band Algorithm, the work bands, evaluation bands and list of work bands \( W \) are also identical to those algorithms. Hence, the number of work bands is \( |W| = \Theta \left( \prod_{i=2}^{d} \frac{k_i}{m} \right) \) as for the Row Algorithm. The evaluation order of the vertices as well as the data that is kept in internal memory is as specified in Theorem 3.13.

Let us determine the maximum size \( m \) of the sweep shape. For each of its \( m^{d-2} \) columns, a sweep shape consists of \( \lceil \frac{m}{B} \rceil \) blocks of data. Hence, the number of vertices contained in the blocks of a sweep shape is

\[
B \cdot \lceil \frac{m}{B} \rceil \cdot m^{d-2} = \Theta \left( m^{d-1} \right).
\]

The algorithm needs to keep at least \( 2s - 1 = \Theta(1) \) complete sweep shapes in internal memory for the input, i.e. the sweep shape currently being evaluated as well as the \( s - 1 \) preceding and \( s - 1 \) proceeding sweep shapes. For the asymptotic analysis we disregard blocks of the output that have to reside in internal memory.\(^5\) Hence, the number of vertices that have to be in internal memory at once is lower bounded by

\[
(2s - 1) \cdot \Theta \left( m^{d-1} \right) = \Theta \left( m^{d-1} \right).
\]

We therefore know that the size of the sweep shape is

\[
m = \Theta \left( \frac{d-1}{\sqrt{M}} \right).
\]

To determine the number of non-compulsory I/Os consider the \( k \)-intersections. As for the Row Algorithm, the 1-intersection of each work band (besides the first and the last work band in each direction) is characterized by \( \Phi(W, 1, h) \) being a \( d-1 \) dimensional hypercube of side length \( (m - 4s) \) lying in the middle of \( W \). Hence per \( H_h \) and work band \( W \) the section above and below the evaluation band in \( x_2 \) direction is a \( d-1 \) dimensional hypercube of \( 2s \) vertices in \( x_2 \) direction and \( (m - 2s) \) for all directions \( x_i \) for \( i \in \{3, \ldots, d\} \). Hence, there are \( (m - 2s)^{d-2} \) columns per work band and \( H_h \) that contain vertices that are also shared with other work bands. This means that there are at least

\[
2 \cdot (m - 2s)^{d-2} = \Theta \left( m^{d-2} \right)
\]

blocks per work band and \( H_h \) causing non-compulsory I/Os. With the conservative assumption that all these blocks are only shared by two work bands, each of them causes one non-compulsory I/O for each work

\(^5\) If the vertices are evaluated in lexicographic order one output block per column of the sweep shape, hence \( m \) blocks in total, should be kept in memory. This can be reduced to one output block if the vertices are evaluated according to columns, i.e. with \( x_2 \) and not \( x_d \) being the fastest changing index for the evaluation order.
Band (see also the similar discussion in the proof of Theorem 3.13). We disregard all other $2 \cdot (d - 2)$ faces of the evaluation band that also contain vertices of the k-intersections for $k \geq 2$. These vertices only increase the number of non-compulsory I/Os but would not change the asymptotic behavior of the leading term of the non-compulsory I/Os.

Hence, the Column Algorithm causes at least

$$\Theta \left( \prod_{i=2}^{d} \frac{k_i}{m} \right) \cdot k_1 \cdot \Theta \left( m^{d-2} \right) = \Theta \left( \prod_{i=1}^{d} \frac{k_i}{m} \right) = \Omega \left( \prod_{i=1}^{d} \frac{k_i}{a-\sqrt{M}} \right)$$

non-compulsory I/Os. Although we do not provide a proof, we claim that this analysis is tight, i.e. the Column Algorithm causes that many non-compulsory I/Os and the effects disregarded only concern lower order terms. For $d = 2$ and $d = 3$ this bound reads

$$d = 2: \quad \Omega \left( \frac{k_1 k_2}{M} \right) \quad \text{and} \quad d = 3: \quad \Omega \left( \frac{k_1 k_2 k_3}{\sqrt{M}} \right).$$

The Column Algorithm causes an asymptotically non-optimal number of non-compulsory I/Os as the column-major data layout forces the algorithm to perform one non-compulsory I/O per column of the sweep shape to load and store the k-intersections for $k \geq 2$. Per column, however, there are only a constant number of $2s$ vertices in the k-intersection above and below the evaluation band (besides for the outermost columns). This forces the Column Algorithm to perform a non-compulsory I/O for each column and each step in $x_1$-direction instead of performing an non-compulsory every $2s/B$ steps in $x_1$-direction and each column.

3.7.2 Upper Bounds for 2 Dimensions

The upper bounds in two dimensions are summarized in Table 3.3 and the layouts depicted in Figure 3.2 and Figure 3.4. The Row and the Column Algorithm are both by a factor of $\Omega (B)$ worse than the optimum. The two dimensional Hypercube Band Algorithm achieves the correct asymptotics but the leading term of the non-compulsory I/Os is by a factor 2 worse than the lower bound. In [Leopold02b] Leopold uses a mixed row/column layout which achieves the same constant at the leading term of the non-compulsory I/Os as the Hypercube Band Algorithm. In contrast, the constant of the lower bound is matched by the Diagonal Band Algorithm which is depicted in the right part of Figure 3.4. The key observation is that shifting a sweep shape which lies in a hyperplane of normal $(1, -1)$ in the two unit directions $(1, 0)$ and $(0, 1)$ alternately doubles the vertices of an evaluation band while the number of vertices of the 2-intersections stays constant per evaluation band.
Table 3.3: Leading term of the non-compulsory I/Os for different algorithms in two dimensions.

<table>
<thead>
<tr>
<th>2D Algorithms</th>
<th>Sweep Shape</th>
<th>Sweep Seq.</th>
<th>Non-Compulsory 1/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Algorithm</td>
<td>Vertical</td>
<td>$e_1$</td>
<td>$\Omega\left(\frac{1}{M} \cdot k_1 k_2\right)$</td>
</tr>
<tr>
<td>Column Algorithm</td>
<td>Vertical</td>
<td>$e_1$</td>
<td>$\Omega\left(\frac{1}{M} \cdot k_1 k_2\right)$</td>
</tr>
<tr>
<td>Hypercube Band Alg.</td>
<td>Vertical</td>
<td>$e_1$</td>
<td>$\frac{8s^2}{BM} \cdot k_1 k_2$</td>
</tr>
<tr>
<td>Diagonal Band Alg.</td>
<td>Diagonal</td>
<td>$e_1, e_2$</td>
<td>$\frac{4s^2}{BM} \cdot k_1 k_2$</td>
</tr>
</tbody>
</table>

The Diagonal Band Algorithm is optimal with respect to the leading term of the non-compulsory I/Os. As suggested by the lower bound, the algorithm evaluates the grid $\ell^1$-ball by $\ell^1$-ball, by sweeping through adjacent $\ell^1$-balls (see Figure 3.5). The algorithm fits the framework of Theorem 3.13 and is depicted in detail in Figure 3.4.

The sweep sequence $\mathcal{X}$ consists of both unit directions, $\mathcal{X} = \{e_1, e_2\}$. The sweep shape $\mathcal{S}$ is a diagonal line segment of $m$ points,

$$\mathcal{S} = \left\{ x \in \mathbb{Z}^2 : (x_1 + x_2 = 0) \land (x_2 \geq 0) \land (x_2 < m) \right\}.$$ 

By definition, $b = 1$ holds. The sweep shape $\mathcal{S}$ can be regarded as the intersection of an $\ell^1$-ball with a hyperplane, i.e. a line, of normal $(1,1)$ through the center of that ball. This means that we are sweeping an $\ell^1$-ball diagonal per diagonal.

The work bands that result from this construction are also diagonal and extend along the direction $(1,1)$ of the sweep. One work band consists of $m$ vertices per diagonal hyperplane of normal $(1,1)$. Using two different shifts for the sweep sequence together with a diagonal sweep shape, however, doubles, in comparison to the Hypercube Band Algorithm, the vertices of a work band per hyperplane $H_h$ to $2m$. 
The evaluation bands are also diagonal bands. They consist of \( m - s \) vertices per hyperplane of normal \((1,1)\) and \( 2m - 2s \) vertices per \( H_h \). Hence, \( |E^\infty_h| = 2m - 2s \) and \( e = 2 \) (Assumption 4). Furthermore, an evaluation band also consists of \( 2m - 2s \) vertices per hyperplane of normal \( e_2 \) and hence Assumption 3 holds. The evaluation bands lie in the middle of the work bands. In particular, consider the intersection of a work band \( W \) with a \( H_h \). First, there are \( s \) vertices which belong to \( W \setminus E_W \), followed by \( 2m - 2s \) vertices of the evaluation band \( E_W \) and another \( s \) vertices of \( W \setminus E_W \). Hence, Assumption 5 is satisfied.

It is easy to cover the grid with non-overlapping evaluation bands as their structure is very simple. Shift the infinite evaluation bands by \( 2m - 2s \) vertices in direction \( e_1 \) until the whole grid is covered (Assumption 2). By this approach a work band overlaps with at most two other work bands, the work bands above and below it, and hence Assumption 7 is satisfied. This covering also immediately yields that at most

\[
\frac{k_1 + k_2}{2m - 2s} \overset{(3.21)}{=} \frac{k_1 + k_2}{2s} - O(B) = O\left(\frac{k_1}{M}\right)
\]
evaluation bands and hence also work bands are needed to cover the grid. Therefore Assumption 6 is satisfied.

Let us now analyze the \( k \)-intersections. By the arrangement of the work bands, all \( k \)-intersections for \( k \geq 3 \) are empty and Assumption 9 is satisfied. For the 2-intersections consider the intersection of a work band \( W \) with a hyperplane \( H_h \) of normal \( e_1 \). We have seen that, per \( H_h \), the vertices of the evaluation band are adjacent to \( s \) vertices of \( W \setminus E_W \) on each side of the evaluation band. Also, the adjacent work bands, intrude by exactly \( s \) vertices into the current work band. Hence, \( 4s \) vertices of the evaluation band, \( 2s \) vertices on each side, are part of the 2-intersections for each \( H_h \). Hence, \( \Phi(W,2,h) = 4s \) and Assumption 8 holds with \( b = 4s \).

We verify the remaining Assumption 1 with Lemma 3.18.

**Lemma 3.18.** Given \( d = 2 \) and the setup of Definition 3.12. Given the Diagonal Band Algorithm specified in this section: the sweep sequence is \( X = \{e_1, e_2\} \), the sweep shape \( \mathcal{S} \) is a diagonal line segment of \( m \) points and the list of work bands \( W \) is as specified in this section. Then, for any \( W \in W \) and any \( w \in E_W \) the following holds:

\[
I(w) = [o_W(w) - \delta, o_W(w) + \delta] \quad \text{with} \quad \delta = s \cdot |\mathcal{S}| + O(1) .
\]

**Proof.** By Equation 3.17: \( I(w) = [o_W(w_{\min}), o_W(w_{\max})] \). Therefore, it is left to determine \( w_{\min} \) and \( w_{\max} \) as well as their distance to \( w \) in the work band order. Let \( \mathcal{S} \) be the sweep shape containing \( w \in E_W \). First consider \( w_{\max} \). This vertex has to be in the sweep shape \( \mathcal{S} \) proceeding \( \mathcal{S} \) by \( s \) shifts. Of all vertices in \( \mathcal{S} \cap S_s(w) \) the vertex \( w + (s,0) \) is the
one of maximum lexicographic order, hence \( w_{\max} = w + (s, 0) \). Denote by \( v \) the vertex \( v \) to which \( w \) is shifted in the next \( s \). It holds that

\[
\|v - w\|_W = s \cdot |\mathcal{S}| \quad \text{and} \quad v \in \mathcal{S}_s \cap S_s(w).
\]

The vertices in \( \mathcal{S}_s \cap S_s(w) \) form a contiguous, diagonal line segment of \( s + 1 \) vertices. Hence, the work band distance between \( v \) and \( w_{\max} \) is bounded by \( s \). Hence

\[
\|w - w_{\max}\|_W \leq \|w - v\|_W + \|v - w_{\max}\|_W \leq s \cdot |\mathcal{S}| + O(1).
\]

Similarly, it can be shown that

\[
w_{\min} = w + (-s, 0) \quad \text{and that} \quad \|w - w_{\min}\|_W \leq s \cdot |\mathcal{S}| + O(1).
\]

Hence, the claim follows.

As all assumptions are satisfied, we can apply Theorem 3.13 to yield an upper bound for the non-compulsory I/Os of the Diagonal Band Algorithm (\( c = 1, e = 2 \) and \( b = 4s \)). The upper bound is

\[
4s^2 \cdot \frac{k_1 k_2}{BM} + O \left( \frac{k_1 k_2}{M^2} \right) + O \left( \frac{k_1}{B} \right).
\]

### 3.7.3 Upper Bounds for 3 Dimensions

The upper bounds in three dimensions are summarized in Table 3.4. In three dimensions, the Hypercube Band Algorithm achieves optimal asymptotic behavior and outperforms the Row Algorithm by \( \Omega \left( \frac{1}{\sqrt{B}} \right) \) and the Column Algorithm by \( \Omega \left( \frac{1}{B} \right) \). Using a two dimensional \( \ell^1 \)-ball (Diamond Band Algorithm) instead of a square as sweep shape improves the leading term of the non-compulsory I/Os by a factor of \( \frac{1}{\sqrt{2}} \) compared to the Hypercube Band Algorithm. The best new upper bound improves this by another factor of \( \frac{1}{\sqrt{3}} \), leaving a gap of \( \sqrt{2} \) to the lower bound (Hexagonal Band Algorithm – \( s \in \{1, 2, 3\} \) only). The Hexagonal Band Algorithm shifts a hexagonal sweep shape, resulting from the intersection of a three dimensional \( \ell^1 \)-ball with a plane of normal \( (1, 1, 1) \), and alternates the three unit shifts.

One reason why the constants of the lower and upper bound do not yet match is the fact that the grid cannot be tiled with \( \ell^1 \)-balls in three dimensions. Hence, imitating the lower bound by tiling the grid with \( \ell^1 \)-balls fails in three dimensions while it was possible in two dimensions.

#### 3.7.3.1 Diamond Band Algorithm

The three dimensional Diamond Band Algorithm improves upon the three dimensional (Hyper-)Cube Band Algorithm by a factor of \( \frac{1}{\sqrt{2}} \). In-
Table 3.4: Leading term of the non-compulsory I/Os for different algorithms in three dimensions. The Hexagonal Band Algorithm is analyzed only for \( s \in \{1, 2, 3\} \).

Instead of a two dimensional hypercube (square) it uses a diamond, i.e. \( \ell^1 \)-ball, as sweep shape. It has been proven in Section 3.5.2 that the \( \ell^1 \)-ball has a better interior to boundary ratio than the cube. Hence, it is also advantageous as sweep shape. Although the Hexagonal Band Algorithm presented in Section 3.7.3.2 further reduces the number of non-compulsory I/Os, the Diamond Band Algorithm maybe better for implementation as the complexity of its data layout is still manageable.

Theorem 3.13 can be used to analyze the Diamond Band Algorithm. The sweep sequence consists of the first unit shift, \( X = \{ e_1 \} \). The sweep shape is a two dimensional \( \ell^1 \)-ball of radius \( m - 1 \), i.e. side length \( m \), lying in a hyperplane of normal \( e_1 \),

\[
S = b_2^{(m-1,0)} = \{ (x_1, x_2, x_3) \in \mathbb{Z}^3 : (x_1 = 0) \land (|x_2| + |x_3| \leq m - 1) \}.
\]

A sweep shape consists of

\[
|S|^\text{Section 3.5.3} = m^2 + (m - 1)^2 = 2m^2 - 2m + 1
\]

vertices and hence \( c = 2 \).

The work and evaluation bands are simple to describe as we are only sweeping in \( x_1 \)-direction:

\[
W^\infty_h = S = b_2^{(m-1,0)} \quad \text{and} \quad E^\infty_h = b_2^{(m-1-s,0)}.
\]

An evaluation band lies in the center of its corresponding work band. In particular, Assumption 3 and Assumption 5 hold.

It follows that

\[
|E^\infty_h| = (m - 1 - s)^2 + (m - 1 - s - 1)^2 = 2m^2 + O(m)
\]

and \( e = 2 \) (Assumption 4).
Sweeping only in $x_1$-direction means that if we cover $[k_2] \times [k_3]$ with $E^{\infty}_h$, then the three dimensional grid is covered by the resulting evaluation bands. In fact, it is easy to cover the two dimensional grid with two dimensional $\ell^1$-balls without overlap (Assumption 2). See Figure 3.6 and also [Bis04] for an example covering. This covering gives rise to the list of work bands $W$. Any work band overlaps only with up to the $8 = 3^d - 1$ work bands adjacent to it and hence Assumption 7 holds.

Let us now consider the $k$-intersections. Pick one work band $W$. The $(E^\infty_W)_h$ is an $\ell^1$-ball of radius $(m - 1 - s)$ and at the same center as the work band $W^\infty_h = \mathcal{S} = b_2^{(m-1,0)}$. Therefore, a work band $W$ is larger than $E_W$ by $s$ layers of vertices. As the evaluation bands cover the grid without overlap, a work band $W' \neq W$ can only reach the $s$ outermost layers of $E_W$. Hence only the outermost $2s$ layers of $W$ may be shared with other work bands. Therefore,

$$|\Phi_{W,2,h}| \leq |\Gamma_{2s}(b_2^{(m-1,0)})| \leq \sum_{r=1}^{2s} 4(m - r) = 8s \cdot m - \Theta(1).$$

This means that Assumption 8 holds with $b = 8s$. It also follows from the shape and position of the evaluation and work bands that the $k$-intersections are limited to constant size per $H_h$ whenever $k \geq 3$. By Lemma 3.14 it follows that $|\Phi_{W,k,h}| = \Theta(1)$ for $k \geq 3$. Hence, also Assumption 9 is satisfied.

The number of work bands used in total is the same as the number of work bands per $H_h$ as we are only sweeping in $x_1$-direction. Hence, as the necessary assumptions are satisfied, we can apply Lemma 3.16 to yield Assumption 6:

$$|W| \leq \frac{(k_2 + \Theta(m))(k_3 + \Theta(m))}{|E^{\infty}_h| - \Theta(m)} = \frac{(k_2 + \Theta(m))(k_3 + \Theta(m))}{(2m^2 + \Theta(m)) - \Theta(m)} \overset{(3.21)}{=} \Theta \left( \frac{k_2k_3}{M} \right).$$

Figure 3.6: How the Diamond Band Algorithm covers the two dimensional grid $[k_2] \times [k_3]$ with $E^{\infty}_h$, i.e. two dimensional $\ell^1$-balls.
Finally, we can apply Lemma 3.17 as we are only sweeping in $x_1$-direction to show that Assumption 1 holds. Therefore all assumptions of Definition 3.12 are met and Theorem 3.13 can be applied. Hence, the number of non-compulsory I/Os of the 3-dimensional Diamond Band Algorithm is upper bounded by $(c = e = 2, b = 8s)$

$$8s^{3/2} \cdot \frac{k_1k_2k_3}{B \cdot \sqrt{M}} + O\left(\frac{k_1k_2k_3}{\sqrt{B \cdot M}}\right).$$

3.7.3.2 Hexagonal Band Algorithm for 3 Dimensions: Alternate Sweeps Improve the Constant

The best algorithm presented in three dimensions is the Hexagonal Band Algorithm. Essential to the Hexagonal Band Algorithm is the concept of alternating shifts which has already proven useful in two dimensions for the Diagonal Band Algorithm. The Hexagonal Band Algorithm is asymptotically optimal and improves the constant of the leading term of the non-compulsory I/Os by a factor of $\sqrt{3}$ over the three dimensional Hypercube Band Algorithm and a factor of $\frac{\sqrt{3}}{\sqrt{2}}$ over the three dimensional Diamond Band Algorithm.

The Hexagonal Band Algorithm is presented to show that alternating shifts are very likely necessary if the lower and upper bounds should match. Using alternating shifts, however, makes the analysis and implementation of the algorithm more difficult. Due to the sloped traversal of the grid and the irregularity of the $k$-intersections implementing this data layout requires sophisticated logic and index computations to determine the $k$-intersections. The overhead created by the irregular $k$-intersections may cancel the performance gained by reducing the non-compulsory I/Os. Hence, it is unsure whether implementing this algorithm is worthwhile. Regarding the analysis, some of the concepts introduced in Section 3.6 are solely necessary to cope with a sweep sequence containing different shifts. As the Hexagonal Band Algorithm is therefore more of theoretical interest, we present its main ideas and concepts but do not proof all details in a rigorous manner. In particular we only analyze this algorithm for $s \in \{1, 2, 3\}$ and claim that the modifications necessary to the sweep shape are identical for all $s' = s + 3 \cdot k$ for $k \in \mathbb{N}_0$. The case distinction between different $s$ is necessary to show that the evaluation bands cover the grid in the desired manner. Also, we only show that the evaluation bands can cover the grid without overlap by presenting a picture and do not explicitly construct the work band list $W$ by giving the offsets between the work bands. In addition, counting the vertices of a work band that cannot be evaluated (see Equation 3.32) and those that are part of $k$-intersections for $k \geq 2$ (see Equation 3.34) is done by proof by picture. We leave it to the reader to verify these details in a rigorous manner for all $m$. Instead, we describe the essential results and build upon the intuition gained from the other algorithms.
Figure 3.7: Hexagonal Band Algorithm. Left: for \( m=3 \), intersecting a 3-dimensional \( \ell^1 \)-ball (black) with plane of normal \((1, 1, 1)\) results in \( \mathcal{S}' \) (red). Additional vertex of \( \mathcal{S} \) for \( s = 1 \) in cyan. Middle: trying to cover the two dimensional grid \([k_2] \times [k_2] \) with the \( \{ w \in W : P_s(w) \subseteq W \}_h \subset E^\infty_h \) resulting from the original hexagonal sweep shapes \( \mathcal{S}' \) for \( m = 3 \) and \( s = 1 \). One vertex (circled) cannot be covered when the \( E^\infty_h \) do not overlap. Right: projection \( P_2 \) of 2-star stencil \( S_2 \). The projection \( P_2 \) of the stencil consist of the different level sets of the original stencil \( S_2 \).

The Hexagonal Band Algorithm is a memory efficient band algorithm. To describe it, fix the sweep sequence \( \mathcal{X} \) to consist of the three unit vectors in their natural order, \( \mathcal{X} = \{ e_1, e_2, e_3 \} \). The sweep shape \( \mathcal{S}' \) of size \( m \) is the intersection of the 3-dimensional \( \ell^1 \)-ball of radius \( 2m \) centered at the origin, \( b'_3(2m, 0)(0) \subset \mathbb{Z}^d \), with the plane \( H \) through the origin and normal \((1, 1, 1)\),

\[
\mathcal{S}' = b'_3(2m, 0)(0) \cap H^{(1,1,1)}_0 \quad \text{(see Figure 3.7)}.
\]

The vertices in \( \mathcal{S}' \) can be counted per \( x_1 \)-level-sets and hence

\[
|\mathcal{S}'| = 2 \cdot \left( \sum_{k=m+1}^{2m} k \right) + (2m + 1) = 3m^2 + 3m + 1.
\]

The sweep shape \( \mathcal{S}' \) has hexagonal structure and it is therefore easy to cover the grid with the work bands. For \( s = 3 \cdot k \), \( k \in \mathbb{N} \), the evaluation bands resulting from the sweep shape of size \( m \) are identical to the work bands resulting from a sweep shape of size \( m - 2k \). Hence, in this case the grid can be covered with non-overlapping evaluation bands without adapting the sweep shape and we can choose

\[
\mathcal{S} = \mathcal{S}' \quad \text{for } s = 3 \quad \text{(3.28)}.
\]

For \( s \notin 3\mathbb{N} \) the evaluation bands, however, do not cover the grid in a nice fashion (see Figure 3.7 for an example). Consider the cases \( s \in \{1, 2\} \). In summary, when the evaluation bands are placed without overlap there is one vertex missing per evaluation band and \( H_h \). If the evaluation bands overlap such that the whole grid is covered the overlap would be of order \( \mathcal{O}(m) \) per \( H_h \) and would affect the leading
Figure 3.8: \( s = 1 \) and \( m = 3 \). The center of \( W^\infty_h \) is marked with a cross. Right: \( W^\infty_h \). Vertices \( w \in W^\infty_h \) that can be evaluated because of \( P_s(w) \subseteq W^\infty_h \) are depicted as circles, vertices for which \( P_s(w) \not\subseteq W^\infty_h \) holds are squares. Vertices of the original sweep shape \( \mathcal{S}' \) in red, additional vertices of the adapted sweep shape \( \mathcal{S} \) in cyan. Middle: a partial, non overlapping cover of the grid \([k_2] \times [k_3]\) with the sets \( \{ \ w \in W^\infty_h : P_s(w) \subseteq W^\infty_h \} \subseteq \mathcal{E}^\infty_h \). Left: Overlapping work bands to estimate the vertices in the \( k \)-intersections.

term of the non-compulsory I/Os. Therefore, the sweep shape \( \mathcal{S}' \) is enlarged by one vertex in a manner that the grid can be covered with non-overlapping evaluation bands. The enlarged sweep shapes \(^6 \mathcal{S} \) are given by

\[
\begin{align*}
\text{s} = 1 : \quad & \mathcal{S} = \mathcal{S}' \cup (-m, -1, (m + 1)) & \text{and} \quad (3.29) \\
\text{s} = 2 : \quad & \mathcal{S} = \mathcal{S}' \cup (-1, -m, (m + 1)) & \text{.} \quad (3.30)
\end{align*}
\]

In any case, \( c = 3 \) holds.

Let us now discuss the evaluation bands and how they cover the grid for \( s \in \{1, 2\} \) in detail. Therefore, consider the intersection of a (infinite) work band with a \( \mathcal{H}_h \). This intersection is depicted in Figure 3.8 for \( s = 1 \) and in Figure 3.9 for \( s = 2 \). The intersection consists of

\[
|W^\infty_h| = 3 \cdot |\mathcal{S}'| = 3 \cdot (3m^2 + 3m + 2)
\]

vertices, as we employ three different unit shifts and hence each subset of \( \mathcal{S} \) of a fixed \( x_1 \)-coordinate appears three times in this intersection.

To determine which vertices of the work band belong to the evaluation band and the \( k \)-intersections, regard how the shifts of the sweep shape affect the \( s \)-star stencil. We define \( P_s(\cdot) \) (see Figure 3.7), the two dimensional projection of the \( s \)-star stencil \( S_s(\cdot) \), as

\[
P_s(w) := \{ v \in (h \times [k_2] \times [k_3]) : \|w - v\|_1 \leq s \} \cup \\
\cup \{ y : \|w - v\|_\infty \leq s \land (w_i \leq 0 \land v_i \leq 0 \quad \text{for} \quad i = 2, 3) \} \cup \\
\cup \{ y : \|w - v\|_\infty \leq s \land (w_i \geq 0 \land v_i \geq 0 \quad \text{for} \quad i = 2, 3) \}
\]

\(^6\)These are not the only possible choices to enlarge \( \mathcal{S} \) but there are up to six different choices for each \( s \).
We will argue next that a \( w \in W \) can be evaluated, if the projection of the \( s \)-star stencil of \( w \) is in \( W \cap H_h \), i.e.

\[
P_s(w) \subset (W \cap H_h) \quad \Rightarrow \quad w \in E_W .
\] (3.31)

If \( P_s(w) \subset (W \cap H_h) \), all vertices of \( P_s(w) \) belong to some sweep shape of \( W \). In particular, every fixed vertex \( v \in P_s(w) \) has a trace in the 3-dimensional grid depending on the sweep shape(s) it belongs to and the shifts that are applied to it and its sweep shape(s). It is important that the whole trace of a vertex belongs to \( W \) if \( v \) itself belongs to \( W \). As the three unit shifts alternate as sweeps, the vertices \( v \pm k \cdot (1,1,1) \) for \( k \in \mathbb{Z}^d \) are in the trace of \( v \) independently of which shift is next. As example consider, the vertex \( v = (-1,-1,0) \) which has the vertex \( (0,0,1) \) in its trace. Verifying the \( \supseteq \) part of the equality

\[
P_s(w) = \bigcup_{k=-s}^{s} \left( (S_s(w) \cap H_k) - k \cdot (1,1,1) \right)
\]

yields that the traces of the vertices of the stencil projection \( P_s(w) \) cover the stencil \( S_s(w) \) itself. Hence \( S_s(w) \subset W \) follows from which \( w \in E_W \) follows.

The structure of \( W_h^\infty \) (see Figure 3.8 and Figure 3.9) can be described as follows: the vertices of \( W_h^\infty \) can be split into groups of three diagonals that all correspond to vertices of the same \( x_1 \)-value of the sweep shape. The three diagonals within a group correspond to the three different unit shifts. Within such a group, the three diagonals have the same number of vertices. If the group contains vertices of \( x_1 \)-value \( u \), then there are \( (2m+1) - |u| \) vertices in the group. The only exception is the group which contains the additional vertex, \( (m,1,-(m+1)) \) for \( s = 1 \)
and \((-1,-m,(m+1))\) for \(s = 2\). This group contains one additional vertex per diagonal, or three additional vertices in total.

To estimate \(E^\infty_h\) apply the projected stencil \(P_s\) to \(W^\infty_h\). By Equation 3.31: if the projected stencil of a vertex \(w\) is in \(W^\infty_h\), then \(w\) itself belongs to the evaluation band. Hence

\[
E \supset \{w \in W : P_s(w) \subseteq W\} \quad \text{and} \quad W \setminus E \subset \{w \in W : P_s(w) \not\subseteq W\}.
\]

The vertices in the set

\[
A := \{w \in W : P_s(w) \not\subseteq W\}
\]

can be counted per \(H_h\). To treat all cases for different \(s\) at once, we apply \(P_s(\cdot)\) to the work band \((W')^\infty\) resulting from the original \(\mathcal{S}'\). If \(P_s(\cdot)\) is within \((W')^\infty\) then it is for sure within \(W^\infty\) and hence this underestimates the number of vertices that can be evaluated. Furthermore, enlarging the sweep shape adds at most three vertices to \(A\). The first and the last \(2s\) diagonals of \(W^\infty_h\) \((m + 1\) vertices each) are in the set \(A\). For each group of three diagonals with \(x_1 \neq 0\), there are \(2 \cdot 2s\) vertices in \(A\) at the ends of the diagonals. This holds also for the first and last group for which we have already excluded several diagonals completely. Again, this double accounting only decreases the vertices in the evaluation band and weakens the analysis. The middle group for \(x_1 = 0\) contains \(2s + 3s\) vertices for which \(P_s(w) \not\subseteq (W')^\infty\) holds. In total (counting diagonal by diagonal from left to right),

\[
|W \setminus E|_h \leq (m + 1) \cdot 2s + m \cdot 4s + 5s + m \cdot 4s + (m + 1) \cdot 2s + 3 = 12ms + O(1).
\]

Hence, a lower bound for the size of \(E^\infty_h\) is given by

\[
|E^\infty_h| \geq 3 \cdot (3m^2 + 3m + 2) - (12ms + O(1)) = 9m^2 - O(m) \quad (3.33)
\]

and \(e = 9\) follows (Assumption 4).

Besides the sheer number of vertices in \(E^\infty_h\), we also need to know the shape of \(E^\infty_h\) to cover the grid with few evaluation bands. Examine the sets \(W^\infty_h\) and

\[
A^\infty_h = \{w \in W : P_s(w) \not\subseteq W\} \cap H_h
\]

depicted in Figure 3.8 and Figure 3.9 for \(m = 3\) and \(s = 1\) respectively \(s = 2\).7 From the hexagonal structure of the sweep shapes follows that the two sets \(W^\infty_h\) and \(A^\infty_h\) also have a hexagonal like structure. Figures 3.8 and Figure 3.9 show partial covers of the grid \([k_2] \times [k_3]\) with non-overlapping sets \(A^\infty_h\) which can be extended to cover the whole grid \([k_2] \times [k_3]\). Hence we can also cover the 3-dimensional grid \([k_1] \times [k_2] \times \]

7 \(E^\infty_h\) can also contain further vertices but a subset of \(E^\infty_h\) is sufficient for the following analysis.
As the sweep shape and also $E^\infty$ are hexagonal it follows that a work band overlaps with at most 6 other work bands and Assumption 7 is satisfied. Furthermore, Assumption 3 holds by construction for the adapted $\mathcal{S}$. Assumption 5 holds for the unmodified sweep shape $\mathcal{S}'$. This is sufficient as Assumption 5 is only needed for Lemma 3.16 estimating the total number of work bands which we apply to the unmodified sweep shape $\mathcal{S}'$ in Equation 3.35.

Let us now check the $k$-intersections. The vertices in the $k$-intersections

$$B := \bigcup_{k=2}^\infty \Phi_k(W,h) = \{w \in W : w \in H_h \text{ and } \exists V \in W : W \neq V \text{ and } w \in V\}$$

can be counted similarly to those in $W \setminus E$ (see Equation 3.32). Again, let us first count the vertices in $B$ for the unmodified work band $(W')^\infty_h$. By construction, the

$$\{w \in W' : P_s(w) \subseteq W'\}$$

(subsets of evaluation bands)

do not overlap. Furthermore, as the $W'_h$ are similar to convex shapes, every vertex in $W'_h$ is in the projected stencil $P_s(\cdot)$ of some vertex

$$\{w \in W' : P_s(w) \not\subseteq W'\}.$$

Hence, work bands neighboring $W'_h$ intrude into

$$\{w \in W' : P_s(w) \subseteq W'\}$$

by at most this projected stencil. Therefore we can account for the vertices of $W'$ that are also part of other work bands similar to Equation 3.32. At most the first and last $2 \cdot 2s$ diagonals of $W'_h$ (at most $m + 1$ vertices each for the original $\mathcal{S}'$) are shared with other work bands. For each group of three diagonals for $x_1 \neq 0$ there are $2 \cdot 2 \cdot 2s$ vertices at the ends of the diagonals that can also belong to other work bands, possibly being identical with the vertices of the first and last diagonals that have already been accounted for completely. The middle group for $x_1 = 0$ contains $2 \cdot 5s$ vertices which are also part of other work bands. When we now consider $\mathcal{S}$ instead of $\mathcal{S}'$ there are at most 3 more vertices in $W_h$ then in $W'_h$. Conservatively, we assume that these three vertices are in $k$-intersections for $k \geq 2$. Similarly, all 6 work bands adjacent to $W$ contain at most 3 more vertices than per $H_h$ than assumed for $\mathcal{S}'$. Hence, at most another $3 \cdot 6$ vertices of $W_h$ belong to $k$-intersections for $k \geq 2$. All in all, the number of vertices in $k$-intersections for $k \geq 2$
is at most (counting diagonal by diagonal from left to right as in Equation 3.32)
\[ \left| \bigcup_{k=2}^{\infty} \Phi_{(W,k,h)} \right| \leq 2 \cdot (12m + \Theta(1)) + (3 + 6 \cdot 3) = 24m + \Theta(1). \] (3.34)

This also gives the bound \( \left| \Phi_{(W,2,h)} \right| \leq 24m + \Theta(1) \) and therefore Assumption 8 is satisfied with \( b = 24s \). Furthermore, looking at the effective placement of the work bands, only \( k \)-intersections for \( k \leq 3 \) are non-empty. Hence, the sets \( \Phi_{(W,k)} \) are empty for \( k \geq 4 \). If \( k = 3 \), every \( k \)-intersection is limited in all three unit directions to constant width within a hyperplane \( H_h \). Hence, any \( k \)-intersection for \( k = 3 \) contains only a constant number of vertices per \( H_h \). Using Lemma 3.14 this yields \( \left| \Phi_{(W,k,h)} \right| = \Theta(1) \) for \( k = 3 \), all work bands \( W \) and all \( h \in [k_1] \). Hence Assumption 9 holds.

Assumption 1 is proven by the following Lemma.

**Lemma 3.19.** Given the setup of Definition 3.12, employ the sweep sequence \( X = \{e_1, e_2, e_3\} \), sweep shape \( \mathcal{S} \) as specified in one of Equation 3.29, Equation 3.30 or Equation 3.28 and list of work bands \( \mathcal{W} \) specified in this section. Then, for any \( W \in \mathcal{W} \) and any \( w \in E_W \) the following equality holds:
\[ I(w) = [o_W(w) - \delta, o_W(w) + \delta] \quad \text{with} \quad \delta = s \cdot |\mathcal{S}| + \Theta(\sqrt{M}). \]

*Proof.* By Equation 3.17: \( I(w) = [o_W(w_{\min}), o_W(w_{\max})] \). Therefore, it is left to determine \( w_{\min} \) and \( w_{\max} \) and their distance to \( w \) in the work band order. By the definition of \( I(w) \) we know that \( w_{\min}, w_{\max} \in S_s(w) \). From the lexicographic order it then follows that
\[ w_{\min} = w + (-s, 0, 0) \quad \text{and} \quad w_{\max} = w + (s, 0, 0). \]

The vertex \( v \) to which \( w \) is shifted in he next \( s \) shifts has distance
\[ \|v - w\|_W = s \cdot |\mathcal{S}| \]
from \( w \) in the work band order. For \( \mathcal{S}_s \), the sweep shape proceeding \( \mathcal{S} \) by \( s \) shifts, consider \( \mathcal{S}_s \cap S_s(w) \). Both \( v \) and \( w_{\max} \) are in \( \mathcal{S}_s \cap S_s(w) \). Consider the projection of the set \( \mathcal{S}_s \cap S_s(w) \) in the \( x_1 x_2 \)-plane. Up to translations, this projection is given by
\[ \left\{ x \in \mathbb{Z}^2 : x_1 \geq 0 \land x_2 \geq 0 \land x_1 + x_2 \leq s \right\}. \]

As all vertices of \( \mathcal{S}_s \cap S_s(w) \) belong to the \( s \)-point stencil \( S_s(w) \) they are distributed over at most \( s \) adjacent rows and at most \( s \) adjacent columns of the sweep shape \( \mathcal{S}_s \). By definition, a row of the projection of the sweep shape contains less than \( (2m + 1) + 1 = \Theta(m) \) vertices. Hence, the positions of the lexicographic minimum and maximum vertices of
$\mathcal{S} \cap S_s(w)$ differ by at most $(2m + 2) \cdot s + s = \Theta(m)$ in the work band order. Hence, also $\|w_{\text{max}} - v\|_W = \Theta(m)$ and therefore

$$\|w - w_{\text{max}}\|_W \leq \|w - v\|_W + \|v - w_{\text{max}}\|_W = s \cdot |\mathcal{S}| + \Theta(m) \quad (3.21) \quad s \cdot |\mathcal{S}| + \Theta(\sqrt{M}) \, .$$

The same argument can be used to show that $\|w - w_{\text{min}}\|_W \leq s \cdot |\mathcal{S}| + \Theta(\sqrt{M})$. Hence, the claim follows.

It is left to estimate the total number of work bands. Identify a start and an end of each work band with respect to the order of the shifts. As all shifts are unit directions, we say that work bands start at the 2 dimensional faces of the grid which contain the point $(0,0,0)$.

These are the 3 sets

$$0 \times [k_2] \times [k_3], \quad [k_1] \times 0 \times [k_3] \quad \text{and} \quad [k_1] \times [k_2] \times 0 \, .$$

The face $0 \times [k_2] \times [k_3]$ of the grid can also be written as $H_0$. Hence, Lemma 3.16 can be applied to give an upper bound for the number of work bands that start in this face of the grid. All other faces of the grid in which work bands start can be treated similarly. Denote by $H_i^0$ the hyperplane of normal $e_i$ and distance $h$ from the origin. Considering the unmodified sweep shape $\mathcal{S}'$, the sweep shape and the work bands are symmetric with respect to the three coordinates $x_1$, $x_2$ and $x_3$, i.e. permuting the coordinates does not affect the sweep shape. In particular, the intersections $(E')^\infty \cap H_i^0$ are identical for all $i \in \{1, 2, 3\}$ up to translations and an isomorphism of the coordinates. As in Equation 3.33,

$$(E')^\infty \cap H_i^0 = 9m^2 - \Theta(m) \, .$$

As the modified sweep shape $\mathcal{S}$ enlarges $\mathcal{S}'$ we get $\mathcal{S}$,

$$\left( (E')^\infty \cap H_i^0 \right) \subset \left( E^\infty \cap H_i^0 \right) \, .$$

Hence, the technique of Lemma 3.16 (first enlarging the sides of grid by $2l_i + 4s$ for last $d - 1$ directions and then dividing the number of vertices in one face of the grid by the number of vertices of $E_{H_i^0}^\infty$) can be applied to every face of the grid to bound the number of work bands which start.

---

8 Similarly, work bands end at the 3 faces of the grid which contain the vertex $(k_1 - 1, k_2 - 1, k_3 - 1)$. 
at this face. All in all, the number of work bands needed to cover the grid with evaluation bands is at most (Assumption 6)

\[
\sum_{j=1}^{3} \left( \prod_{i \neq j}^{3} (k_i + O(m)) \right) \left( \frac{\prod_{i=1}^{2} (k_i + O(m))}{(9m^2 - O(m)) - O(m)} \right) = 3 \cdot \frac{\prod_{i=1}^{2} (k_i + O(m))}{9m^2 - O(m)} = O \left( \frac{k_1 k_2}{M} \right). \tag{3.35}
\]

As all assumptions of Definition 3.12 are satisfied, Theorem 3.13 gives an upper bound for the number of non-compulsory I/Os performed by the Hexagonal Band Algorithm \((c = 3, e = 9\) and \(b = 24s)\),

\[
\frac{8\sqrt{2}s^n/2}{\sqrt{3}} \cdot \frac{k_1 k_2 k_3}{B \cdot \sqrt{M}} + O \left( \frac{k_1 k_2 k_3}{\sqrt{B} \cdot M^2} \right).
\]

3.8 Discussion on variants of the theoretical model

This section discusses variants of the theoretical model that are closer to real caches and the effects of these model upon the algorithms and lower bounds.

The main contribution of this chapter are the lower bounds as they do not only provide part of the complexity result but actually guided the construction of the data layouts and algorithms improving the upper bounds. The theoretical model chosen explicitly manages the cache, assumes simple I/Os and counts read and write operations. Furthermore, the given task assumes that we do not work in-place and considers stencil operations to be atomic. We now examine the consequences of dropping these assumptions to model scenarios closer to current hardware. In short, besides working not-in-place none of these assumptions is crucial and matching bounds in one model translate to matching bounds in the other models.

Let us first drop the assumption that I/Os are simple and consider non-simple I/Os, i.e. copy a block of data from external to internal memory when it is accessed. For the moment still assume that the cache is managed explicitly, i.e. we can decide what happens if a block is evicted from internal memory (either store the block back to external memory or simply delete it) and we can write to blocks of the external memory before loading them first. For the upper bounds, non-simple I/Os would mean that the number of compulsory I/Os stays constant while the leading term of the non-compulsory I/Os halves. I/Os regarding the output are either part of the compulsory term or lower order terms. For the input, we never need to store the input values back to external memory as the original copy remains unchanged. As a non-compulsory read in the model using simple I/Os was always preceded
by one non-compulsory write, the leading term of the non-compulsory I/Os in the non-simple model halves. The same argument holds for the lower bounds. Therefore, consider a vertex and the series of I/Os it causes. In the simple model, the first compulsory read of an input vertex is followed by a series of alternating non-compulsory writes and reads, ending with a read. Hence, also for the lower bounds the number of compulsory I/Os stays constant at $2 \cdot \prod_{i=1}^{d} k_i$ while the number of non-compulsory I/Os halves when switching from simple to non-simple I/Os. Hence, dropping the assumption that I/Os are simple reduces the number of non-compulsory I/Os by a factor of 2 for both the lower and the upper bounds.

Switching from an explicit to an implicit cache affects the cache replacement strategy as well as how data is evicted from internal memory and the access to blocks we only want to store data in. Regarding the cache replacement strategy, assuming an implicit LRU cache replacement strategy instead of managing the cache replacement strategy explicitly does neither affect the lower nor the upper bounds (except possibly the hexagonal band algorithm). An explicit cache replacement strategy can be seen as an optimal cache replacement strategy and hence lower bounds for an explicitly managed cache replacement strategy hold for any implicit strategy. The upper bounds have already assumed a LRU strategy for all input blocks. For the output blocks and all algorithms besides the hexagonal band algorithm it is easy to see that there also is at least one vertex per k-intersection and sweep shape. Hence, a global LRU strategy never evicts an unfinished output block (except when the end of the current work band is reached).

A block that is evicted from internal memory can either be stored back to external memory or deleted within internal memory and forgotten. In an implicit cache it is common to write the block to external memory, causing a write operation, only if it has been altered in internal memory. Hence, if the block has not been modified in internal memory, and is hence identical to the external memory copy, the block is deleted within internal memory saving the I/O. For the algorithm as well as the lower bound this is exactly the behavior we employed for non-simple I/Os in the explicit model and hence the bounds do not change.

Finally in an implicit cache, a block has to be loaded to internal memory before results can be stored in it. This increases the number of compulsory I/Os to 3 times the number of grid points, as we need to read the input as well as all output blocks and store the output as well. It does, however, not change the leading term of the non-compulsory I/Os as I/Os caused by the output either affect the compulsory I/Os or lower order terms of the non-compulsory I/Os. Hence, for the lower as well as the upper bounds in an implicit cache model using non-simple I/Os the number of compulsory I/Os increases to $3 \cdot \prod_{i=1}^{d} k_i$ while the leading
term of the non-compulsory I/Os remains unchanged with respect to
the explicit model using non-simple I/Os.

If we would just be interested in reads and disregard the write opera-
tions in an implicit cache using non-simple I/Os, the compulsory term
goes back down to 2 I/Os per grid point for both the lower and the
upper bounds. Also, the non-compulsory term would be unchanged, as
the leading term of the non-compulsory I/Os results from reading the
2-intersections of the input grid. These vertices are never written and
hence counting only reads does not change the term. The additional
reads only amount to lower order terms as they are just caused by the
first and the last block of each k-intersections of the output grid.

As next step, let us consider to work in-place in an implicit model us-
ing non-simple I/Os and counting reads only. By working in-place, the
compulsory read of the output grid can be avoided and the number of
compulsory read operations drops to 1 per grid point. The presented
algorithms, however, are not capable to work completely in-place. Some k-
intersections for $k \geq 2$ need to be buffered in additional space in external
memory as they are needed to evaluate vertices after the k-intersection
itself has been evaluated. A naïve buffer for all vertices in the union
of all k-intersections for $k \geq 2$ would require $O\left(\left(\prod_{i=1}^{d} k_i\right) / d^{-\sqrt{M}}\right)$
additional space. This can be reduced to $O\left(\left(\prod_{i=1}^{d-1} k_i\right) / d^{-\sqrt{M}}\right)$ by
working through adjacent work bands first and reusing the buffer of
one k-intersection when all the work bands it is part of have been eval-
uated. When the input values of a k-intersection for $k \geq 2$ are stored to
the buffer when the k-intersection itself is evaluated, the number of non-
compulsory reads a vertex of that k-intersection takes part in rises from
$k - 1$ to $k$. This additional non-compulsory read is caused by loading
the block of the buffer into which we want to store the input data. In
the worst case, for $k = 2$, this means that the number of non-compulsory
I/Os doubles and hence the leading term of the non-compulsory I/Os
doubles. The lower bound transfers to the in-place setting, as working
in-place is more restrictive. It is likely that the lower bound can be im-
proved for the in-place setting such that previously matching lower and
upper bounds match again.

Last but not least, let us address the assumption that evaluating a
stencil is an atomic operation which cannot be split. Allowing partial
evaluations of the stencil requires a more general lower bound while the
upper bounds still apply. Furthermore, although partial stencil evalu-
ations are possible in practice, none of the implementations discussed
in the related work takes advantage of partial evaluations of the stencil.
Regarding the lower bounds, we think the assumption that stencil oper-
ations are atomic can be dropped without weakening the lower bounds.
Given a set of vertices which we want to evaluate in one round of the
algorithm, the isoperimetric inequalities yield how many grid points
need to be transferred to (or have already been transferred from) other
rounds. This does not assume that the stencil is indivisible but only states that neighboring values are needed to evaluate the stencil. Reducing the number of vertices that need to be transferred from one round to another would mean to compress the data which has to be disallowed for the EM model to make sense.

3.9 conclusions

Stencil computations are basic building blocks for many applications from scientific computing, including finite difference methods used to solve PDEs. This work has considered one update of the grid according to the s-star stencil in the external and parallel external memory model. As the constant of the leading term of I/Os can be easily matched by a simply blocking strategy, this chapter has analyzed the constant of the second term, i.e., the constant of the leading term of the non-compulsory I/Os. For that, new algorithms were derived and data layouts that exactly match the access patterns of these algorithms were proposed. The lower bounds were improved by carefully applying an isoperimetric result to the rounds into which an arbitrary algorithm can be split. In combination, the gaps that existed between the lower and the upper bounds since 2002 were reduced significantly. In particular, the new bounds match in two dimensions and close a multiplicative gap of 4.

An experimental consideration on how to turn the proposed algorithms into high performance code remains open. Although memory access is very important for high performance code, it is not the only factor influencing its runtime. It needs to be determined in a process of algorithms engineering if, and to which extent, the benefits from an optimized data layout and an optimized data access lead to faster code. Other options that may influence runtime include the more complicated index computations, optimizations for several layers of the memory hierarchy, vectorization and loop unrolling enabling scalar replacement.

Topics the chapter does not address include a time step setup, matching bounds for non-trivial B and standard layouts like a usual row or column layout, stencils different from star stencils and matching bounds for high dimensions. The lower bounds have not been tuned to account for different data layouts as this would change the isoperimetric inequalities. However, while accounting for a specific data layout further restricts the theoretical model it may be a key aspect to get matching lower and upper bounds for different layouts. It would also be interesting to examine the I/O complexity of stencils different from the star stencils given by $\ell^1$-balls. Canonical candidates are stencils described by $\ell^\infty$-balls and mixtures between $\ell^1$- and $\ell^\infty$-stencils appearing in finite element methods. It also remains open if the complexity can be pinpointed for three and higher dimensions as both, lower and upper bounds, do not seem optimal yet.
This chapter did not consider a time step, i.e., updating the whole grid several times according to the stencil, as a time step introduces a directed dimension and hence changes the structure of the computation graph, the stencil defining the neighborhood of a set and hence the isoperimetric sets and inequalities. The best upper bounds in two (Diagonal Band Algorithm) and three dimensions (Hexagonal Band Algorithm), however, should easily be transferable to a setting where one of the spatial dimensions is replaced by a temporal one. The structure of these algorithms is compatible with the setting of one temporal and one respectively two spatial dimensions. Parallelizing the algorithms is, however, more difficult when there is a temporal dimension. To transfer the lower bounds to a time step setting an isoperimetric inequality for the directed, multi-layer, time step computation graph needs to be derived. With this new inequality, the rest of the argument can be applied as before. In general, in a setting with time step, the number of computations is the product of the spatial and temporal dimensions whereas the number of compulsory I/Os solely depends on the spatial dimensions. This implies, that the non-compulsory term dominates the complexity and the optimizations and lower bounds address the leading term of the I/Os in a time step setting. As a result higher speedups are expected from the optimizations and the optimizations become more likely to be used in practice.
4 BACKGROUND ON SPARSE GRIDS

4.1 INTRODUCTION AND MOTIVATION

Solving high-dimensional numerical problems, in particular high-dimensional PDEs, is one of the grand challenges of current and future high-performance computing systems. A PDE is considered high-dimensional if it is posed in more than the classical four dimensions, space plus time. A prominent example stems from plasma fusion [GLB+11]. The simulation of hot plasmas in fusion reactors of Tokamak or Stellerator type [HBB+12] is either based on a gyrokinetic approach or on the Vlasov-Maxwell equations [GLB+11]. Both PDEs have to jointly treat, besides time, three spatial dimensions and two or three, respectively, velocity dimensions. Employing higher-order time stepping schemes, in each time step a five- or six-dimensional PDE has to be solved.

Unfortunately, straightforward discretization schemes, as full grids, fully suffer the “curse of dimensionality”, a term going back to Bellman in the 60s [Bel61]. The curse of dimensionality describes the fact that the number of unknowns of the discretization grows exponentially with the dimensionality. Hence, high-dimensional applications are among the compute-hungry drivers of exascale computing [DB+11]. Consider a discretization with 1024 grid points in one dimension. This results in $1024^d$ grid points in $d$ dimensions. For $d = 5$ we obtain more than $10^{15}$ unknowns and require more than 8 PetaBytes to only store that many values in double precision. Hence, simply because of the massive amount of data, high-dimensional problems are out of scope for computation with classical discretizations. Typical simulations have to restrict themselves to small subsets of the domain. In case of the international flagship project ITER, a Tokamak-style fusion reactor built in Caderache, France, it is expected that at least $10^{11}$ degrees of freedom, i.e. grid points, for about $10^6$ time steps are necessary for a global simulation with sufficient accuracy.

Fortunately, hierarchical discretization schemes are of help: given certain smoothness conditions, a discretization based on sparse grids – a term coined by Zenger [Zen91] for the solution of high-dimensional PDEs – can reduce the number of grid points by 8 orders of magnitude

---

1 As this setting includes boundary points, there would be precisely 1025 grid points per dimension.
from $10^{15}$ to $10^7$ in the example above. In general, sparse grids lessen the curse of dimensionality

$$O\left(h_n^{-d}\right) \quad \text{to} \quad O\left(h_n^{-1} \cdot |\log_2 h_n|^{d-1}\right)$$ (4.1)

for dimension $d$ and minimum mesh size $h_n = 2^{-n}$. Sparse grids reduce the number of unknowns by first performing a base change from the full grid basis to the hierarchical basis and then selecting the most important basis function of the hierarchical basis. The hierarchical basis, depicted in Figure 4.1, can discriminate between basis functions as the area of support of the basis functions shrinks as the grid is refined further. A tensor product approach is used to generate the high-dimensional basis functions from the depicted one-dimensional basis functions. As the area of support of the basis functions is problem independent, the sparse grid can select the most important basis functions, i.e. those that have a large area of support, a priori. By doing so, sparse grids reduce the degrees of freedom significantly while the accuracy of the solution only deteriorates slightly for sufficiently smooth functions [BG04]. This reduction of the degrees of freedom enables the solution of higher dimensional problems.

In particular, sparse grids have been used to solve a variety of high-dimensional numerical problems, including PDEs from fluid mechanics [GHZ96], financial mathematics [Hol11, BHPS11] and physics [KH13], real-time visualization applications [BPB11, BJPM12], machine learning problems [Pfl10], data mining problems [GG01, BPZ08] and so forth.

While sparse grids lessen the curse of dimensionality, they are not able to break or avoid the curse completely as can be seen in Equation 4.1. The number of grid points still depends exponentially on the dimension. The base, however, reduces significantly. This has enabled the successful application of sparse grids without boundary points to problems with up to a couple of hundred of dimensions. When boundary points need to be included, however, sparse grids are usually limited to around 10 dimensions as the constants hidden in the $O$-notation are important then. Keep in mind that this is still a huge improvement compared to the starting point of regular grids for which even 5 dimensional problems are barely feasible.

The base change from the full grid basis to the hierarchical basis is crucial for sparse grids. It is called hierarchization and is one of the most fundamental algorithms for sparse grids.

Whereas the hierarchical basis enables a lessening of the curse of dimensionality, it also introduces more complicated data dependencies to non-neighboring grid points (see Figure 1.3). In the full grid basis, depicted in Figure 4.1, basis functions do only overlap if they correspond to directly neighboring grid points. In the hierarchical basis, a basis function overlaps with all of its hierarchical predecessors and descendants.
4.1 Introduction and Motivation

Therefore, numerical problems on sparse grids are harder to solve as the algorithms have to account for the altered data dependencies. As a result not a lot of algorithms have been implemented to work directly in the hierarchical basis of sparse grids. Those that have, take advantage of the unidirectional principle.

The unidirectional principle is the enabling algorithmic principle regarding sparse grids. Instead of working in $d$-dimensional space, the problem is decomposed into $d$ distinct phases each dealing with 1-dimensional subproblems, called poles [Jac14], of the current work dimension. This breaks the complicated $d$-dimensional data dependencies of the sparse grid. Besides simplifying the algorithms themselves, the unidirectional principle also reduces the flop count as intermediate results can be propagated through the $d$ phases. Any unidirectional algorithm is, however, inherently cache inefficient as it it performs $d$ sweeps over the data. Hence, the number of cache misses any unidirectional algorithm has to cause is at least (let $C_\ell$ denote the input grid.)

$$d \cdot \frac{|C_\ell|}{B} - (d - 1) \cdot \frac{M}{B} = d \cdot \frac{1}{B} \left( |C_\ell| - \frac{d - 1}{d} \cdot M \right).$$ (4.2)

In this equation, the term $(d - 1) \cdot M/B$ describes the fact that at most an entire memory of $M/B$ blocks can be transferred between two consecutive phases of the unidirectional principle. This work refers to Equation 4.2 as the unidirectional lower bound for the number of cache misses.

Instead of working directly in the hierarchical basis of sparse grids, there is another common approach to sparse grids: the sparse grid combination technique [GSZ92] depicted in Figure 1.4. The sparse grid combination technique, or simply combination technique, is an extrapolation scheme. It solves the original problem formulation for many, coarse anisotropic regular grids also called component grids [Heg03a]. A suitable linear combination of the component grid solutions then retrieves a single solution in the hierarchical sparse grid space. These smaller
subproblems on the component grids can be solved independently and therefore in parallel. The hierarchical approach thus overcomes a central problem of massively parallel computations: the splitting into the component grids ensures scalability on future high-performance computers by breaking the global communication requirements of conventional discretization approaches. Furthermore, employing the combination technique there is no need to change the application code if it can be applied to arbitrary anisotropic and regular grids. In the case of the plasma physics simulations, the code GENE [GLB^+11] is employed in a current project [KPJH12, HHK^+] within the German priority program “Software for exascale computing”. Note that the combination technique can additionally be used to ensure further requirements for next-generation computing such as fault tolerance (see Section 4.2).

While the sparse grid combination technique breaks the global communication requirements, communication is still necessary. After every or at least after few time steps [GHSZ93, GHZ96] of a time dependent PDE, the solutions of the component grids should be combined to the sparse grid solution (reduce) and the joint solution distributed back to the component grids (broadcast). This introduces a synchronization barrier which requires a shrunken but global communication as illustrated in Figure 4.2. The reduce/broadcast step remains as the main communication bottleneck of the sparse grid combination technique.

To take advantage of the hierarchical structure of sparse grids in this remaining communication step, the component grid solutions need to be hierarchized, i.e. transferred from the full grid basis to the hierarchical sparse grid basis. For communication schemes that rely on the hierarchical representation of the component grid solutions, hierarchization is on the performance critical path as shown in Figure 4.2.
Therefore, Chapter 5 first presents a memory efficient implementation of the unidirectional hierarchization algorithm for component grids. As this implementation almost achieves the minimum runtime for unidirectional hierarchization algorithms, the question arises: can we design a hierarchization algorithm that avoids the unidirectional principle, the dominating design pattern for sparse grids, and beats the unidirectional lower bound for the number of cache misses? Chapter 6 answers this question successfully with a divide and conquer hierarchization algorithm for isotropic component grids $C_\ell$ and, in addition, complements this algorithm with a lower bound for the leading term of the non-compulsory cache misses. Having memory efficient hierarchization algorithms at hand, Chapter 7 addresses the reduce/broadcast step of the combination technique by deriving two optimal communication schemes for this remaining communication bottleneck.

This work analyzes algorithms for the component grids of the sparse grid combination technique. If and how the algorithmic ideas can also be applied to regular or adaptive sparse grids is discussed in the chapter respective to the algorithmic idea. Also, only piecewise linear basis functions as those depicted in Figure 4.1 are discussed. Hierarchization for other basis functions is very similar but can require more floating point operations. While Chapter 5 designs the unidirectional hierarchization algorithm for component grids without boundary, the divide and conquer hierarchization algorithm of Chapter 6 and the communication schemes for the reduce/broadcast step of Chapter 7 address component grids with boundary. This has solely historic reasons and it would be simple to modify the unidirectional hierarchization algorithm such that it can handle component grids with boundary points.

As usual in the sparse grid literature, we assume that the dimension $d$ is constant and hence omit it in the $O$-notation. This has three main reasons: first, the dimension is problem specific and does not grow as we refine the grid to obtain a more accurate solution. Second, employing the $O$-notation with two independent variables lets the analysis explode as fewer terms dominate each other. Third, although sparse grids allow to tackle higher-dimensional problems, the dimensions stay moderate. In particular when boundary points are included, the dimension rarely exceeds 10. When no boundary points are used, then the constant of the dimension may play a significant role. Hence, we use a compromise when analyzing the divide and conquer hierarchization algorithm of Chapter 6: the dimension $d$ is assumed to be constant and the complexities of the algorithm and the lower bound are once listed omitting this constant and once stating it explicitly. The unidirectional hierarchization algorithm of Chapter 5 is not analyzed theoretically but experimentally and no asymptotic analysis is used in that chapter. The communication schemes for the reduce/broadcast step of the combination technique are analyzed theoretically as well as experimentally in Chapter 7. As the
experiments take all effects into account, we assume that \( d \) is constant when we state the complexity results in the \( O \)-notation.

In general, we represent vectors in bold face and denote by \( \| \cdot \|_1 \) the \( \ell^1 \)-norm. The index \( r \) is always from the set \( r \in \{1, \ldots, d\} \). Relations and functions on vectors are used component wise, e.g.

\[
\ell \leq k \iff \ell_r \leq k_r \quad \text{and} \quad \ell < k \iff (\ell_r \leq k_r \quad \forall r \quad \text{and} \quad \exists r \quad \text{s.t.} \quad \ell_r < k_r)
\]

As well, \( 2^{-\ell} = (2^{-\ell_1}, \ldots, 2^{-\ell_d}) \). Furthermore, we assume that the component grids contain boundary points and the boundary of the sparse grid is refined as the interior of the grid, i.e. level 1 boundary is used for the sparse grid (see Section 4.4).

The rest of this chapter is organized as follows: Section 4.2 discusses related work regarding sparse grids. Then, Section 4.3 introduces basic notation regarding full grid spaces and hierarchical increment spaces as well as their corresponding grids. Subsequently, the different ways to allow sparse grids to represent functions with non-zero values at the boundary are discussed in Section 4.4. Section 4.5 introduces the unidirectional principle. The concept of the hierarchical predecessors is crucial for sparse grids, in particular for the hierarchization algorithm, and therefore discussed in Section 4.6. Then the hierarchization task (Section 4.7) as well as the unidirectional hierarchization algorithm (Section 4.8) are described. This chapter concludes with Section 4.9 giving the formulation of the hierarchization algorithm as stencil computation.

For further details and background on sparse grids refer to the survey from Bungartz and Griebel [BG04].

**Research Contributions.** This chapter introduces the necessary background and notation to derive and analyze algorithms on sparse grids. Its contents have been presented with a different foci in the following publications and ongoing projects upon which this work builds: [Hup14b, Hup13, HJH+14, HJ14a, HHJP14]. These projects were and are conducted jointly with Riko Jacob, Markus Hegland, Mario Heene and Dirk Pflüger and this chapter merges the relevant background and notation from the various manuscripts. Therefore, this chapter contains text parts from all mentioned manuscripts. While this chapter contains contributions from different co-authors up to and including Section 4.3, the text is based upon my contributions from Section 4.6 onwards. Hence, parts of this chapter up to and including Section 4.3 might also appear in the PhD thesis of Mario Heene.

### 4.2 Related Work

The term sparse grids was coined by Zenger [Zen91] for the solution of PDEs using a hierarchical function space decomposition. Some of the underlying ideas even date back to Smolyak [Smo63]. Refer to
for a survey on sparse grids. Shortly after sparse grids where first used to solve PDEs the so-called sparse grid combination technique was proposed by Griebel, Schneider and Zenger [GSZ92]. Sparse grids and the sparse grid combination technique have been continuously improved ever since. Due to the underlying hierarchical approach, adaptive refinement is straightforward when discretizing and treating high-dimensional problems directly in the sparse grid space [Pfl12]. The sparse grid combination technique can be refined, too, but in a slightly more restricted, dimensionally adaptive way adding complete anisotropic grids [Heg03a]. Furthermore, the combination coefficients of the combination technique can be chosen in a problem-dependent, optimal manner [Heg03b, HGC07].

Working with the combination technique instead of directly in the sparse grid’s hierarchical basis has further pros and cons: The advantages include the easy parallelization of the combination technique on the coarse component grid level [Gri92, GHR92, GHZ96, GG01, GHN03, BP12, HKP14] and the ability to use standard solvers on the component grids. Furthermore, the redundancy given by the component grids allows one to build algorithm-based fault tolerance [HA84] into the combination technique [HH13, LHH+13, HH14]. In particular the additional coarse grain level of parallelism and the ability to incorporate algorithm-based fault tolerance, make the combination technique suitable for high-performance computing, even on the exascale level. Drawbacks of the combination technique include the need to assemble the sparse grid solution from the component grids solutions and hence the need for a reduced but global communication scheme to do so. For time-dependent problems, it has been recognized that assembling the sparse grid solution is necessary after few time steps [GHSZ93, GHZ96]. Also, when the different component grids are distributed over a network of compute nodes, load balancing becomes an issue. The runtime of computing the solution of a problem on one component grid does not only depend on the number of unknowns of the component grid but also on its degree of anisotropy [HKP14].

Early communication schemes used a farm of slaves to compute the component grid solutions and then assembled the sparse grid on a master [Gri92, GHR92, GHZ96]. This imposes a 1 to p bottleneck as all p slaves have to communicate with the master. For three dimensions, a scheme reducing this bottleneck to \( O(\sqrt{p}) \) by splitting the communication into 1-dimensional communication tasks was also proposed but not implemented [GHSZ93]. Also, all-to-all communication has been used to assemble the sparse grid solution on all component grids [Kra02].

Hierarchization describes the base change from the full grid basis to the hierarchical basis and is as such one of the fundamental and most important algorithms for sparse grids. Furthermore, one of the communication schemes of Chapter 7 relies on the representation of the com-
ponent grids in the hierarchical basis and hence on hierarchization as preprocessing step. Dehierarchization describes the inverse basis transformation from the hierarchical basis to the full grid basis. Many implementations of the hierarchization and dehierarchization algorithms exist [Pfl10, MWB+11, MBP+12, BJPM12, Jac14]. All these implementations take advantage of the unidirectional principle. Furthermore, all these approaches focus on hierarchization of regular or adaptive sparse grids which inherent a complicated structure and on which navigation is tedious.

Let us first address the hierarchization algorithms. The standard software for sparse grids is $\text{SG}++$ [Pfl10] and focuses on spatially adaptive sparse grids [Pfl12]. It handles those using level-index vectors as keys in a hash table to index grid points. $\text{SG}++$ is able to hierarchize the anisotropic component grids as a special case.

$\text{CDS}$ [MWB+11] presents a bijective mapping from the set of level-index vectors to a set of contiguous integers. By doing so, it can store the sparse grids points without any memory overhead. In particular, $\text{CDS}$ stores hierarchical increment spaces in contiguous memory. The hierarchization algorithm navigates on the data layout using the bijective mapping between the data in contiguous memory and the level-index vectors. $\text{CDS}$ also performs extensive comparisons with prior software which it all outperforms due to the increased locality of its data storage. $\text{fastSG}$ [MBP+12] extends the bijective mapping of $\text{CDS}$ to truncated grids which cover the component grids of the combination technique as special case.

$\text{rSG}$ [Jac14] efficiently hierarchizes regular sparse grids using a dynamic data layout. It is in particular efficient for high dimensions, up to thousands, and small levels. A work phase hierarchizes the data with respect to the first dimension and a rotate phase rearranges the data such that role of the dimensions change. These two operators are applied alternatingly $d$ times to yield the $d$-dimensional hierarchization. Because of the rotation operation, $\text{rSG}$ only has to hierarchize the grid points with respect to the first dimension. In particular, the poles of the current work dimension are always in contiguous memory. Furthermore, the navigation on the contiguous, 1-dimensional poles is implemented using an iterator and thus avoiding the computation of the level-index vector.

$\text{StructuredSG}$ [BJPM12] applies the idea of a dynamic memory layout to the data structure of $\text{CDS}$. The basic contiguous units of $\text{CDS}$, the hierarchical increment spaces, are rotated to increase locality and enable vectorization. As $\text{StructuredSG}$ is based on $\text{CDS}$ it uses the level-index vector to navigate on the data layout. $\text{StructuredSG}$ outperforms $\text{CDS}$ and can also hierarchize the anisotropic grids of the combination technique. Thus, it provides the fastest hierarchization algorithm for component
grids up to date and is therefore used to benchmark the implementation of Chapter 5.

Efficient evaluation schemes for sparse grids, preliminary for dehierar-
chization, the inverse basis transformation, are addressed by CDS, fastSG
and StructuredSG. Furthermore, CDS and StructuredSG include GPU im-
plementations.

None of these software packages specializes in the anisotropic compo-
nent grids of the combination technique as Chapter 5 does. Hence, also
none of them can fully exploit the regular structure of the component
grids. Also, the recursive, divide and conquer hierarchization algorithm
of Chapter 6 describes the first approach to avoid the unidirectional prin-
ciple globally.

Consider the use of these algorithms as pre- and postprocessing steps
for the communication step of the combination technique as shown in
Figure 4.2. CDS [MWB+11] stores the sparse grid organized by hier-
archical increment spaces, i.e., in the same way as the communication
scheme Subspace Reduce derived in Chapter 7 is going to processes the
data. While CDS cannot hierarchize component grids, its successors
fastSG and StructuredSG can. Hence, when one of these schemes would
be used for pre- and postprocessing, the data would not need to be
reorganized between pre- and postprocessing, i.e., hierarchization and
dehierarchization, and communication. The full grid solver applied be-
tween communication steps, however, is likely to be built for a stan-
dard column- or row-major layout. Hence, the data would need to be
reorganized before hierarchization and after dehierarchization. If the
hierarchization algorithm is designed to work in the same data layout
as the full grid solver, the same reorganization is required but between
(de-)hierarchization and communication. The implementation of the uni-
directional hierarchization algorithm of Chapter 5 works on a standard,
but padded, row-major layout and hence, potentially, on the same data
layout as the full grid solver.

4.3 FUNCTION SPACES AND GRIDS

The representation of a high-dimensional function can be based on a
sparse grid discretization in the hierarchical sparse grid basis which
in turn is based on the hierarchical increment spaces $W_\ell$. In addition,
the combination technique can be used to decompose the whole prob-
lem into several problems on the component grids, which can be com-
puted independently. This makes it necessary to also represent the high-
dimensional function via a linear combination of partial solutions based
on the anisotropic component grids and their respective anisotropic full
grid spaces $V_\ell$. Hence, we first introduce the anisotropic component
grids $C_\ell$ and the corresponding anisotropic full grid spaces $V_\ell$ as well
as the hierarchical grids $H_\ell$ and the respective hierarchical increment spaces $W_\ell$.

Let us begin with a conventional discretization of the $d$-dimensional space $\Omega := [0,1]^d$. For that, use an anisotropic grid $C_\ell$ with mesh-width $h_\ell_r := 2^{-\ell_r}$ and discretization level $\ell_r$ in dimension $r \in \{1, \ldots, d\}$. The grid points $x$ of $C_\ell$ are

$$C_\ell = \left\{ x = \frac{1}{2^\ell} \cdot i \in \Omega : i_r \in \{0,1,\ldots,2^{\ell_r}\} \right\}.$$  

Hence, the grid $C_\ell \subset [0,1]^d$ is completely defined by its level vector $\ell \in \mathbb{N}_0^d$ describing how often dimensions $r \in \{1, \ldots, d\}$ has been refined. A grid of refinement level $l_r$ consist of $2^{l_r} + 1$ grid points in dimension $r$, the outermost two of which, i.e., the points with $i_r \in \{0,2^{l_r}\}$, are boundary points. If $l_r = 1$, the component grid consists of three grid points in dimension $r$. Each grid point can be described completely by its level-index vector pair $(\ell, i)$ by

$$x_{\ell,i} := i \cdot h_\ell = \left( i_1 \cdot 2^{-\ell_1}, \ldots, i_d \cdot 2^{-1_d} \right).$$

In the nodal point of view all grid points of $C_\ell$ are associated with the same level.

Now consider a conventional anisotropic tensor product space $V_\ell$. An anisotropic tensor product space is spanned by $d$-dimensional local tensor product basis functions

$$\varphi_{\ell,i}(x) := \prod_{r=1}^d \varphi_{\ell_r, i_r}(x_r)$$

associated to the corresponding grid points $x_{\ell,i} \in C_\ell$. For simplicity, think of piecewise $d$-linear functions and the classical full grid nodal basis as sketched in the left part of Figure 4.1. Formally, this basis is given by the one-dimensional hat basis functions

$$\varphi_{\ell_r, i_r}(x_r) = \varphi \left( x_r \cdot 2^{\ell_r} - i_r \right) \quad \text{for} \quad \varphi(x) = \max \left( 1 - \|x\|, 0 \right).$$

In general, the tensor product space $V_\ell$ itself is given by

$$V_\ell = \text{span} \{ \varphi_{\ell,i} : x_{\ell,i} \in C_\ell \}.$$  

A function $f_\ell \in V_\ell$ can be represented by

$$f_\ell = \sum_{x_{\ell,i} \in C_\ell} \beta_{\ell,i} \cdot \varphi_{\ell,i}.$$
The coefficients $\beta_{\ell,i}$ are called full grid coefficients. Typically, the basis functions of regular grids are chosen such that they are 1 at they grid point they correspond to and 0 for all other grid points. Hence,

$$\beta_{\ell,i} = f_{\ell}\left(x_{\ell,i}\right). \quad (4.3)$$

The $\beta_{\ell,i}$ are also called nodal coefficients, nodal values, or values at the grid points. As example consider again the full grid nodal basis depicted in the left part of Figure 4.1.

Alternatively, we can represent $V_\ell$ by a unique decomposition into hierarchical increment spaces $W_{\ell'}$ as

$$V_\ell = \bigoplus_{\ell' \leq \ell} W_{\ell'}.$$  

This formulation can be used as definition for the hierarchical increment spaces. For an alternative definition, consider the corresponding grids. In the hierarchical view, the grid $C_\ell$ is split into the hierarchical increment grids

$$H_{\ell} := \left\{ x_{\ell,i} \in C_\ell : \begin{cases} \text{if } \ell_r \geq 1 : & 1 \leq i_r \leq 2^{\ell_r} - 1 \text{ and } i_r \text{ odd} \\ \text{if } \ell_r = 0 : & i_r \in \{0,1\} \end{cases} \right\}.$$  

The grid $C_\ell$ is then given by the disjoint union

$$C_\ell = \bigsqcup_{k \leq \ell} H_k.$$  

The hierarchical increment space $W_\ell$ is spanned by all basis functions $\varphi_{\ell,i}$ which are associated with grid points in $H_\ell$,

$$W_\ell = \text{span} \left\{ \varphi_{\ell,i} : x_{\ell,i} \in H_\ell \right\}.$$  

Denote the coefficients $\alpha_{\ell,i}$ for a certain $w_\ell \in W_\ell$, i.e., the coefficients $\alpha_{\ell,i}$ for which

$$\sum_{x_{\ell,i} \in H_\ell} \alpha_{\ell,i} \cdot \varphi_{\ell,i} = w_\ell \in W_\ell,$$

as (hierarchical) surpluses. We can represent each $f_\ell \in V_\ell$ uniquely as

$$f_\ell(x) = \sum_{k \leq \ell} w_k(x) = \sum_{k \leq \ell} \sum_{x_{k,i} \in H_k} \alpha_{k,i} \cdot \varphi_{k,i} \quad \text{with } w_k \in W_k. \quad (4.4)$$

Figure 4.1 (right) shows (hierarchical) increment grids and hierarchical piecewise linear basis functions corresponding to the hierarchical decomposition. The algorithm that calculates the hierarchical surpluses $\alpha_{\ell,i}$ from the full grid coefficients $\beta_{\ell,i}$ is called hierarchization.
Consider your favorite problem that can be solved (or approximated) in the full grid space \( V_\ell \). One of the easiest problems to think of is the interpolation of a function \( f \) on the grid points of \( \mathbb{P}_\ell \), but the problem may also be as complicated as your favorite PDE. We are interested in the full grid solution \( u_\ell \in V_\ell \) with \( \ell_1 = \cdots = \ell_d = n \) for some uniform discretization level \( n \), which we cannot afford. But for sufficiently smooth functions, the sparse grid function \( f_n \in V_n^{SG} \),

\[
V_n^{SG} := \bigoplus_{\|\ell\|_1 \leq n + d - 1} W_\ell,
\]

yields an accurate approximation with orders of magnitudes less degrees of freedom by a careful a priori selection of hierarchical increment spaces. The grid points of the sparse grid of level \( n \) and dimension \( d \), \( \text{SG}_n^d \), are given by

\[
\text{SG}_n^d = \bigcup_{\|\ell\|_1 \leq n + d - 1} \mathbb{H}_\ell.
\]

The combination technique solves the problem at hand in the sparse grid space \( V_n^{SG} \) by a linear combination of solutions on coarse and anisotropic regular grids. These full, but anisotropic and coarse grids \( \mathbb{P}_\ell \) are called component grids. Relevant for the combination technique are the component grids \( \mathbb{P}_\ell \) for \( n \leq \|\ell\|_1 \leq n + d - 1 \).

Assume that \( f_\ell \) is the solution of the problem at hand in the full grid space \( V_\ell \). The combination technique solution \( f_n^{CT} \in V_n^{SG} \) is then given by

\[
f_n^{CT} = \sum_{r=0}^{d-1} (-1)^r \binom{d-1}{r} \sum_{\|\ell\|_1 = n + d - 1 - r} f_\ell, \quad f_\ell \in V_\ell.
\]  

(4.5)

In particular, the union of all grids \( \mathbb{P}_\ell \) employed in this combination results in the grid points of the respective sparse grid as shown in Figure 1.4, i.e.,

\[
\text{SG}_n^d = \bigcup_{n \leq \|\ell\|_1 \leq n + d - 1} \mathbb{P}_\ell.
\]

Any function \( f_\ell \in V_\ell \) on the component grid \( \mathbb{P}_\ell \) is uniquely represented by its full grid or hierarchical coefficients. All we require for either hierarchization or the communication task of the combination technique are the vectors of their coefficients. We use the terms “grid points” and “coefficient vectors” equivalently as each grid point corresponds to exactly one basis function and thus to one coefficient. If we refer to communicating a set of grid points the corresponding coefficients are meant.
The subsequent discussion is therefore restricted to the respective grids instead of function spaces and the terminology is used less strict: we may call $H_\ell$ an hierarchical increment space although we refer to hierarchical increment grid $H_\ell$ and its grid points. Let us summarize and define:

- $C_\ell$: the grid, i.e., component grid, corresponding to the anisotropic full grid space $V_\ell$,
- $H_\ell$: the grid corresponding to the hierarchical increment space $W_\ell$,
- $SG_n^d$: the grid corresponding to the sparse grid space $V_{n}^{SG}$ in $d$ dimensions,
- $C_n^d$: the set of all component grids in Equation 4.5 for a sparse grid of level $n$ in $d$ dimensions, $C_n^d := \{C_\ell : n \leq \|\ell\|_1 \leq n + d - 1\}$, and
- $H_n^d$: the set of all hierarchical grids for $V_{n}^{SG}$ in $d$ dimensions, $H_n^d := \{H_\ell : \|\ell\|_1 \leq n + d - 1\}$.

$|C_\ell|$ denotes the number of grid points while $|C_n^d|$ counts the number of component grids.

**4.4 Boundary Treatment**

While there is only one common approach to equip full grids, as the component grids, with boundary points, there are three different ways for sparse grids. The approaches differ with respect to the amount of grid points spent for the boundary treatment. The existing functions can be modified to extrapolate towards the boundary, new basis functions can be added to existing increment spaces or new increment spaces can be created to host the new basis functions. See Figure 4.3 for the respective modified 1-dimensional hierarchical basis functions. While this work focuses on the third case in which boundary functions are added to existing hierarchical increment spaces (level 1 boundary), this section describes all three types of boundary functions for sparse grids.

Let us first discuss the component grids $C_\ell$. As a component grid is already a full, regular grid, refining the boundary in the same way as the interior of the grid is natural. Hence, a component grid with boundary consists of the points

$$C_\ell = \left\{ \frac{1}{2^{\ell}} \cdot i \in \Omega : i_r \in \{0, 1, \ldots, 2^\ell - 1\} \right\}.$$

while a component grid without boundary points is given by

$$C_\ell = \left\{ \frac{1}{2^{\ell}} \cdot i \in \Omega : i_r \in \{1, \ldots, 2^\ell - 1\} \right\}.$$
As the component grids are already full grids, adding boundary points does not increase the total number of grid points significantly.

Let us now address the boundary treatment for sparse grids. The first approach modifies, for each level, the outermost basis functions of the basic one-dimensional setting of the tensor product approach such that the new basis functions extrapolate towards the boundary. A sparse grid with basis functions extrapolating towards the boundary has no explicit boundary basis functions and, accordingly, can be treated like a sparse grid without boundary. The advantage of an extrapolating boundary treatment is that no additional degrees of freedom, i.e. grid points, are necessary. This allows sparse grids with extrapolating boundary to represent function spaces with up to a couple of hundred of dimensions while allowing non-zero boundary values.

In the second approach, new hierarchical increment spaces are created host the additional boundary basis functions. The new boundary basis functions are added on a new level 0 as depicted in Figure 4.4. Accordingly, this boundary is also called level 0 boundary. As a result, the size of SGd as well as the size of all component grids changes as they all now include boundary points. The size of the former increment grids \( H_n \) does not change, however. Only new increment grids are added as the level vector \( \ell \) can now also attain the value 0. Therefore, this approach changes the sets of hierarchical increment grids \( H_n \) as well as the set of component grids \( C_n \). This means that additional increment grids as well as component grids with min(\( \ell \)) = 0 are added to these sets. As the boundary basis functions are added on the additional level 0, the boundary is refined once further than the interior of the sparse grid. This makes a level 0 boundary very expensive with respect to the number of additional grid points. Almost all grid points are located at the boundary which makes this approach infeasible for higher dimensions.

The third approach is a compromise between the former two. It refines the boundary of the sparse grid as much as the interior and is common when boundary points need to be included explicitly. It allows to equip sparse grids with explicit boundary points as long as the dimensions stay moderate. This second approach adds new boundary functions to existing hierarchical increment spaces. In detail, the new boundary basis functions are assigned to level 1 and hence this kind of boundary is called level 1 boundary. While the level of the boundary basis functions is going to be relevant for the communication schemes of Chapter 7, the boundary basis functions still have level 0 with respect to hierarchical predecessor relation discussed in Section 4.6 and depicted in Figure 4.4. A level 1 boundary does not change the set of increment grids \( H_n \). Only the increment grids \( H_\ell \) themselves change. Whenever min(\( \ell \)) = 1, then the increment grid \( H_\ell \) contains additional boundary basis functions and thus has a larger size. Similarly,
4.5 THE UNIDIRECTIONAL PRINCIPLE AND POLES

The unidirectional principle is the enabling algorithmic principle regarding sparse grids. Instead of working in d-dimensional space the problem is decomposed into d distinct phases each dealing with 1-dimensional subproblems. This breaks the complicated d-dimensional data dependencies of the sparse grid (see Figure 1.3). We call these one-dimensional subproblems poles. A pole in dimension r consists of all points of the grid which only differ in the r.th component, i.e. lie on a line.
parallel to the \( r \)-th coordinate axis. Formally, define the projection \( \pi_r \) in dimension \( r \) of a point \( x \in [0, 1]^d \) as

\[
\pi_r : [0, 1]^d \to ([0, 1]^{r-1} \times \{0\} \times [0, 1]^{d-r}) \simeq [0, 1]^{d-1}
\]

\[
\pi_r : x \mapsto (x_1, \ldots, x_{r-1}, 0, x_{r+1}, \ldots, x_d)
\]

The projection \( \pi_r \) naturally generalizes to a set of points \( S \subset [0, 1]^d \) and to the domain \( \mathbb{Z}^d \). The pole \( (x_{\ell,i}) \) of grid point \( x_{\ell,i} \in C_\ell \) in dimension \( r \) are all points of \( C_\ell \) which project onto the same point as \( x_{\ell,i} \).

\[
pole_r(x_{\ell,i}) := \{ y \in C_\ell : \pi_r(y) = \pi_r(x_{\ell,i}) \}
\]

In general, we say that an algorithm implements the unidirectional principle, if it can be decomposed into \( d \) phases such that within phase number \( r \) only grids points within the same pole of dimension \( r \) are used to update each other. Working on the 1-dimensional poles iteratively for all dimensions allows the unidirectional principle to propagate contributions from grid point \( a \) to \( b \) although there is no pole that contains both \( a \) and \( b \) and hence \( a \) and \( b \) never exchange their information directly. By propagating updates through the different phases, the unidirectional principle can reduce the computational complexity of the problem as we discuss in Section 4.9.

4.6 HIERARCHICAL PREDECESSORS

To state the hierarchization algorithm, we need the concept of the hierarchical predecessor. While only a single basis function attained a non-zero value at any given grid point in the nodal basis this is no longer the case for the hierarchical basis. By definition, we still have \( \varphi_{\ell,i}(x_{\ell,i}) = 1 \). For the hierarchical basis it also holds that for every \( k < \ell \) there exists a \( j \) such that \( \varphi_{k,j} \) is also non-zero at \( x_{\ell,i} \). Hence, all these basis function influence the value \( f_\ell(x_{\ell,i}) \) and all those \( \varphi_{k,j} \) are called indirect hierarchical predecessors of \( \varphi_{\ell,i} \) (or \( x_{\ell,i} \)). If we perform the hierarchization bottom-up, i.e., from large \( \ell \) to small \( \ell \), only the (direct) hierarchical predecessors are going to be relevant, though. To state those, consider the simple form of a level-index vector.

We say that the level-index vector \( (\ell, i) \) of the grid point \( x_{\ell,i} \in C_\ell \) is of simple form if there exists no \( (k, j) \) such that the grid points \( x_{\ell,i} \) and \( x_{k,j} \) have the same coordinates while \( k \leq \ell \) and \( k_r < \ell_r \) for some dimension \( r \). For example, the one-dimensional level-index vectors \((3, 4), (2, 2)\) and \((1, 1)\) all describe the point \( \frac{1}{2} \) but only \((1, 1)\) is of simple form. Bringing a level-index vector into simple form is called reducing it (as for fractions). In general, if any component of the index vector is even, the corresponding level-index vector can be reduced. If all entries of the index vector are odd, then the level-index vector is of simple form.
Reducing a level-index vector can only decrease the level component-wise.

With the reduced level-index vector we can state the (direct) hierarchical predecessors of grid point \( x_{\ell, i} \). Let \((k, j)\) be the reduced form of \((\ell, i)\) and let \(e_r\) denote the standard unit vector in dimension \(r\). The two (direct) hierarchical predecessors of grid point \( x_{\ell, i} \) in dimension \(r\) are \( x_{k, (j-e_r)} \) and \( x_{k, (j+e_r)} \).

One way to think about the hierarchical predecessors is to depict the 1-dimensional poles (say in dimension \(r\)) as the nodes of a complete binary tree in in-order traversal as shown in Figure 4.4. Let us first consider the case without boundary depicted on the left side. If the root of the binary tree is assigned level 1, then the level of a vertex in the binary tree is identical with the reduced level of the corresponding grid point in dimension \(r\), i.e., \(\ell_r\). We say a vertex is on a level \(k_r\) of a pole if its reduced level in dimension \(r\) and therefore also its level in the binary tree is \(k_r\). The left (right) hierarchical predecessor of grid point \( x_{\ell, i} \) is on the left (right) side of \( x_{\ell, i} \) and with respect to the in-order traversal the closest predecessor of \( x_{\ell, i} \) in the binary tree. When no boundary points are used, the second hierarchical predecessor does not exist for the outermost grid points of each (refinement) level.

If the component grid contains boundary points, the two outermost points of the pole are given level 0 and the binary tree is constructed ignoring these two nodes. In addition to the relations of the binary tree, the left boundary point corresponding to index \(i_r = 0\) is also a hierarchical predecessor for all grid points \( x_{k, j} \) of the pole with \(k_r \geq 1\) and \(j_r = 1\). Similarly, the right boundary point corresponding to the level-index vector with \(\ell_r = 0\) and \(i_r = 1\) is also a hierarchical predecessor for all grid points \( x_{k, j} \) of the pole with \(k_r \geq 1\) and \(j_r = 2^{k_r} - 1\).

Both hierarchical predecessors in dimension \(r\) exist if the component grid contains boundary points and the reduced level is at least 1, i.e., \(\ell_r \geq 1\). Furthermore, the reduced level of the hierarchical predecessors of \( x_{\ell, i} \) in dimension \(r\) is strictly smaller than the reduced level of \( x_{\ell, i} \).
in this dimension. For all other dimensions, their reduced levels (and indices) agree.

We call the grid points that are hierarchical predecessors of \( x_{\ell,i} \) with respect to any dimension \( r \) its direct hierarchical predecessors. We get the indirect hierarchical predecessors of \( x_{\ell,i} \) if we apply the hierarchical predecessor relation repeatedly, i.e., the indirect hierarchical predecessors of \( x_{\ell,i} \) are its hierarchical predecessors, the hierarchical predecessors of the hierarchical predecessors, and so forth. The descendants of a grid point \( x_{\ell,i} \) are all grid points that have \( x_{\ell,i} \) as an indirect hierarchical predecessor.

4.7 THE HIERARCHIZATION TASK

Hierarchization describes the base change from the full grid basis to the hierarchical basis and hierarchization is therefore one of the most basic tasks for sparse grids. Let \( f : \Omega \to \mathbb{R} \) be a function. As the focus of this work is the sparse grid combination technique, assume \( f = f_{\ell} \in V_{\ell} \). To project an arbitrary function \( f \) to \( V_{\ell} \), we can simply sample \( f \) at the grid points of \( C_{\ell} \) to get the coefficients \( \beta_{\ell,i} \) of Equation 4.3. The input of the hierarchization task are these coefficients \( \beta_{\ell,i} \). The task is to calculate the coefficients of \( f_{\ell} \) in the hierarchical basis, i.e., the hierarchical surpluses \( \alpha_{\ell,i} \) as given by Equation 4.4. This definition does not use any properties of the component grid. Therefore, it can also be used to define the hierarchization task for a function from a regular or adaptive sparse grid space.

4.8 THE UNIDIRECTIONAL HIERARCHIZATION ALGORITHM

4.8.1 The Unidirectional Hierarchization Algorithm for Component Grids \( C_{\ell} \)

The unidirectional hierarchization algorithm presented in Chapter 5 is tuned for component grids and implemented in C++. Accordingly, the component grid \( C_{\ell} \) is stored in row-major order. Therefore, let \( g \) be the global index of the grid points \( x \in C_{\ell} \) assuming that the grid is stored in row-major order. Furthermore, let

\[
\text{leftPredecessor}(g, r) \quad \text{and} \quad \text{rightPredecessor}(g, r)
\]

denote the global index of the left respectively right hierarchical predecessor of \( x_{g} \) in dimension \( r \). Here, as well as in the following algorithms, \( x \) (as well as \( x_{g} \) and \( x_{\ell,i} \)) stand for the coefficient currently stored at the grid point \( x \).

With this notation, Algorithm 4.1 states the unidirectional hierarchization algorithm for the piecewise linear basis. The outer loop iterates over the \( d \) dimensions and constitutes the unidirectional principle. The inner
The unidirectional hierarchization algorithm

**Algorithm 4.1** The Unidirectional Hierarchization Algorithm for a \(d\)-dimensional component grid \(C_\ell\) of level vector \((\ell_1, \ell_2, \ldots, \ell_d)\).

```
for \(r \leftarrow 1\) to \(d\) do  // unidirectional loop over dimensions
    for each 1-dim pole \(P\) in direction \(r\) do
        for \(k_r \leftarrow \ell_r\) down to \(1\) do
            for all \(x_g\) on level \(k_r\) of pole \(P\) do
                \(x_g = x_g - 0.5 \times x_{\text{leftPredecessor}}(g, r)\)
                \(x_g = x_g - 0.5 \times x_{\text{rightPredecessor}}(g, r)\)
            end for
        end for
    end for
end for
```

three loops update the whole data set once. Hereby, the second loop splits the data set into the 1-dimensional *poles* in the current work dimension \(r\). The inner two loops perform the updates on these 1-dimensional poles bottom up in a *daxpy* like fashion. Within these two loops, the outer one goes through the pole level by level and bottom up, from large \(k_r\) to small \(k_r\). The innermost loop iterates over all grid points of the pole that have the specified level.

The bottom up traversal within the pole ensures that we only need to access the direct hierarchical predecessors in the current dimension \(r\). The indirect hierarchical predecessors in the current dimension \(r\) are not relevant. This is due to two facts: first, all hierarchical predecessors in dimension \(r\) have not been altered yet and can hence be regarded as represented in the nodal basis with respect to the current dimension \(r\). Second, the direct hierarchical predecessors are the closest hierarchical predecessors on the left respective right side of the current grid point in dimension \(r\). As the function spanned by the hierarchical predecessors is piecewise linear between the hierarchical predecessors, the value of the function at the current grid point is therefore determined solely by the value of the direct hierarchical predecessors.

If a node possesses only one hierarchical predecessor, the instruction of **Algorithm 4.1** referring to the non-existent predecessor is ignored. Furthermore, if the component grid contains no boundary points, vertices of level 1 never have hierarchical predecessors and hence it is sufficient if the third loop goes down to level 2 instead of level 1. **Algorithm 4.1** does not specify how to calculate the start of the next pole, how to loop over a certain level of a pole and how to compute the hierarchical predecessors on purpose. These details are given in **Section 5.3.1** discussing the implementation.

Updating the grid points bottom-up has another advantage: the hierarchization algorithm can work in-place as basis functions of a larger refinement level \(\ell\) are 0 at all grid points of a lower, coarser level \(k < \ell\) and hence do not influence them.
4.8.2  The Unidirectional Hierarchization Algorithm for Subsets of Grids

In Chapter 6, we want to hierarchize certain sets $S \subseteq C_\ell$ with Algorithm 4.1 to build a divide and conquer hierarchization algorithm. Therefore, let us generalize the unidirectional algorithm for component grids. We want to compute the hierarchical surpluses of all points of $S$ while we can access and alter the values of a superset $T \supset S$. To perform the update of a grid point $x \in S$ according to Algorithm 4.1, it is required that all direct hierarchical predecessors of $x$ are in $T$. Hence, this is a necessary condition for the generalized unidirectional hierarchization algorithm. The hierarchical surplus can be computed correctly for sure if all indirect hierarchical predecessors (i.e. predecessors of predecessors and so forth) are in $T$. This would mean, however, that $T$ has to contain at least parts of the global boundary of the grid. We are going to see in the proof of Theorem 6.3 that less is required. For the sets $S$ we consider it is sufficient if $T$ contains all hierarchical predecessors of $S$ with respect to the $3^d$-stencil. (The $3^d$-stencil for hierarchization which is going to be defined in Section 4.9). Note that we explicitly do not require that the grid points $T \setminus S$ are hierarchized correctly.

To state the generalized unidirectional hierarchization algorithm we introduce more notation. The grid points of pole $(y_{k,j})$ restricted to a set $T \subset C_\ell$ are

$$\text{pole}_r(y_{k,j}, T) := \text{pole}_r(y_{k,j}) \cap T.$$  

Hence, $\text{pole}_r(y_{k,j}, C_\ell) = \text{pole}_r(y_{k,j})$. The vertices of the restricted pole in dimension $r$ that have level $c$ in dimension $r$ are

$$\text{pole}_r(y_{k,j}, T, c) := \{x_{\ell,i} \in \text{pole}_r(y_{k,j}, T) \land \ell_r = c \text{ for } (\ell, i) \text{ in simple form}\}.$$  

The maximum level in dimension $r$ of a set of grid points $S \subset C_\ell$ is given by

$$\text{maxLevel}_r(S) := \max \{\ell_r : x_{\ell,i} \in S \text{ and } (\ell, i) \text{ in simple form}\}.$$  

With that notation, Algorithm 4.2 gives the unidirectional hierarchization algorithm generalized to sets $T$. As $\pi_r(T)$ contains one grid point of $T$ per pole of dimension $r$ we can use this set $\pi_r(T)$ to describe the second loop of Algorithm 4.1. The maximum level of any grid point of the current pole is determined by $\text{maxLevel}_r(\text{pole}(y_{k,j}, T))$. Finally, the fourth loop updates all elements of $\text{pole}_r(y_{k,j}, T, c)$. Similarly to Algorithm 4.1, also Algorithm 4.2 ignores an instruction if it refers to a hierarchical predecessor not in $T$. If $T = C_\ell$, then Algorithm 4.2 and Algorithm 4.1 are identical.
Algorithm 4.2 The generalized unidirectional hierarchization algorithm for a set \( T \). All level-index vectors are assumed to be in simple form.

\[
\text{function } \text{unidirHierarchize}(T) \\
\quad // \text{unidirectional loop over dimensions} \\
\quad \text{for } r \leftarrow 1 \text{ to } d \text{ do} \\
\quad \quad // \text{loop over all poles in dimension } r \\
\quad \quad \text{for all } y_{k,j} \in \pi_r(T) \text{ do} \\
\quad \quad \quad // \text{update pole bottom up} \\
\quad \quad \quad \text{for } c \leftarrow \text{maxLevel}_r\left(\text{pole}_r\left(y_{k,j}, T\right)\right) \text{ down to } 1 \text{ do} \\
\quad \quad \quad \quad \text{for all } x_{\ell,i} \in \text{pole}_r\left(y_{k,j}, T, c\right) \text{ do} \\
\quad \quad \quad \quad \quad // (\ell,i) \text{ has to be in simple form.} \\
\quad \quad \quad \quad \quad // \text{In particular, } \ell_r = c. \\
\quad \quad \quad \quad \quad // \text{As well, } \ell_s = k_s \text{ for } s \neq r. \\
\quad \quad \quad \quad \quad x_{\ell,i} = x_{\ell,i} - 0.5 \cdot x_{\ell,(i-e_r)} \\
\quad \quad \quad \quad \quad x_{\ell,i} = x_{\ell,i} - 0.5 \cdot x_{\ell,(i+e_r)} \\
\quad \quad \quad \end{end for} \\
\quad \quad \end for \\
\quad \end for \\
\quad \end for \\
\quad \end function
\]

4.9 Hierarchization as Stencil – Direct Hierarchization

It can be observed from either the unidirectional hierarchization algorithm for component grids (Algorithm 4.1) or sets (Algorithm 4.2): assuming \((\ell,i)\) is in simple form, the hierarchical surplus \(\alpha_{\ell,i}\) of the point \(x_{\ell,i}\) is given by the following sum of the initial, nodal values \(\beta_{k,j}\),

\[
\alpha_{\ell,i} = \sum_{a \in \{-1,0,1\}^d} c_a \cdot \beta_{\ell,(i+a)} \quad \text{for} \quad c_a = \left(-\frac{1}{2}\right)^{\|a\|_1}. \quad (4.6)
\]

\(a_r\) is forced to be 0 if \(i_r = 0\) or \(i_r = 2\ell_r\).

If \(a \neq 0\) in Equation 4.6, then \(x_{\ell,(i+a)}\) is a (possibly transitive, i.e. predecessor of a predecessor) hierarchical predecessor of \(x_{\ell,i}\). Assume that \((\ell,i)\) is already in simple form while \((\ell',i')\) denotes the simple form of \((\ell,i+a)\). As bringing the level-index vector into simple form can only decrease the level-vector it holds that \(\ell' \leq \ell\). As \(a \neq 0\), it also holds that there is a dimension \(r\) such that \(\ell'_r < \ell_r\). In particular, to calculate the hierarchical surplus of a grid point only grid points whose level is strictly smaller are required beside the grid point itself. We use this observation in Section 6.3.2 to build an (acyclic) dependency graph \(H\) for hierarchization. Ordering the grid points by their level-sum \(\|\ell\|_1\) gives a
topological sorting of this graph and is essential for the lower bound for the hierarchization task presented in Chapter 6.

Furthermore, we can derive from Equation 4.6 that hierarchization is a stencil-like computation. To calculate the hierarchical surplus $\alpha_{\ell,i}$ of a grid point the stencil

$$
\begin{bmatrix}
-\frac{1}{2} & 1 & -\frac{1}{2} \\
\end{bmatrix}^d
$$

is applied to the nodal values $\beta_{k,j}$. The coefficient 1 corresponds to the center point of the stencil, i.e. the grid point $x_{\ell,i}$ itself. The left and right neighbors in the stencil in dimension $r$ are the left and right hierarchical predecessors of $x_{\ell,i}$ in dimension $r$. In contrast to full grids, these grid points are not direct neighbors of $x_{\ell,i}$. The hierarchical predecessors of $x_{\ell,i}$ are at stride $2^{-\ell_r}$ from $x_{\ell,i}$ in dimension $r$ (if $(\ell, i)$ is in simple form). The stride at which the hierarchical predecessors are away from a certain grid point is hence different for all grid points and dimensions. This implies that also the stencil is different for each grid point: while having the same non-zero coefficients, these coefficients are at a different strides from the center point of the stencil. To give an example how the exponent in Equation 4.7 should be applied to the 1-dimensional stencil, consider the 2-dimensional hierarchization stencil:

$$
\begin{bmatrix}
+\frac{1}{4} & -\frac{1}{2} & +\frac{1}{4} \\
-\frac{1}{2} & +1 & -\frac{1}{2} \\
+\frac{1}{4} & -\frac{1}{2} & +\frac{1}{4} \\
\end{bmatrix}
$$

The hierarchical predecessors of $x_{\ell,i}$ that are relevant in Equation 4.6 and hence appear in the stencil are called the hierarchical predecessors of $x_{\ell,i}$ with respect to the $3^d$-stencil. If $(\ell, i)$ is in reduced form, these are the grid points $x_{\ell,(i+a)}$ for $a \in \left(\{-1,0,1\}^d \setminus \{0\}^d\right)$.

Using the hierarchization stencil to calculate the hierarchical surpluses yields a direct hierarchization algorithm that completely avoids the unidirectional principle. If we apply the stencil bottom-up by decreasing level-sum of the level vector of the grid points, the updates can be performed in-place as only hierarchical predecessors of smaller level sums are used to update a grid point.

The formulation of the hierarchization task as stencil further shows how hierarchization can be formulated as sparse-matrix vector multiplication. The (input) vector stores the nodal values $\beta_{k,j}$ and is of size $R|C_\ell|$. The sparse matrix $M \in R^{|C_\ell| \times |C_\ell|}$ is given by the stencil of Equation 4.7, and has non-zero entries for all grid points the stencil accesses. Hence, there are about $3^d$ non-zero values per row of the matrix. To be precise, if $x_{\ell,i}$ is in the interior of $C_\ell$ then there are exactly $3^d$ non-zero entries
per row of $M$. If $x_{\ell,i}$ is on the boundary of the grid with respect to $s$ dimensions, then there are $3^{d-s}$ non-zero entries per row of $M$.

The drawbacks of such a direct algorithm include that it exploits almost no spatial locality as the grid points of a fixed level sum are distributed evenly over the whole grid. Hence, this direct algorithm would load the whole grid several times for each level-sum to have access to all points of the stencil.

Furthermore, such a direct hierarchization algorithm performs $c \cdot 3^d$ flops per grid point (for some $1 \leq c \leq 2$) as $3^d$ different nodal values $\beta_{k,j}$ are needed to calculate the hierarchical surplus $\alpha_{\ell,i}$. The unidirectional hierarchization algorithm, in contrast, performs only about $3^d$ floating point operations per grid point. The unidirectional algorithm decomposes the $d$-dimensional stencil of Equation 4.7 into its 1-dimensional parts $[-\frac{1}{2} \ 1 \ -\frac{1}{2}]$, and each 1-dimensional part contains just 3 entries. The tensor product structure of sparse grids enables the reconstruction of the $d$-dimensional stencil from the 1-dimensional stencils by applying them dimension by dimension. Building up the $d$-dimensional stencil step by step enables the unidirectional principle to reduce the number of floating point operations.
Hierarchization describes the base change from the full grid basis to the hierarchical basis. As the hierarchical basis is one of the key components that enables sparse grids to reduce the curse of dimensionality, hierarchization is one of the crucial sparse grid algorithms. Furthermore, the hierarchization algorithm is also prototypical for sparse grid algorithms and often the first to be analyzed and optimized. The idea is to try optimizations for the hierarchization algorithm and, if they work, apply them to a wider range of sparse grid algorithms. Furthermore, the combination technique works with regular grids, on one side, which, on the other side, implicitly represent a sparse grid. In particular, when the combination technique is applied to a time dependent problem, hierarchization itself can be an important and regular preprocessing step. Whenever both, a full grid algorithm working in the full grid basis, and a sparse grid algorithm working in the hierarchical basis, are used, hierarchization or the inverse base change need to performed between the application of the two algorithms. This is illustrated in Figure 4.2 for the communication step of the combination technique and a communication scheme that works in the hierarchical basis. Hence, the combination technique and its component grids are one setting for which the hierarchization algorithm is important.

So far, almost all sparse grid algorithms employ the unidirectional principle. In particular, all implementations of the hierarchization algorithm known to the author (see Section 4.2) use the unidirectional principle. Some of these implementations can hierarchize the component grids of the combination technique. Neither of them, however, specializes in the component grids and can hence exploit their structure in comparison with regular or adaptive sparse grids.

The implementation of the unidirectional hierarchization algorithm \emph{combiHier} presented in this chapter is tuned specifically for the component grids of the combination technique. By exploiting the additional structure of component grids in comparison with regular or adaptive sparse grids, \emph{combiHier} comes within a factor of 1.5 of the runtime achievable for large grids by any hierarchization algorithm implementing the unidirectional principle. The implementation is able to outperform the currently fastest generic software \emph{StructuredSG} \cite{StructuredSG} by a factor between 5.8 and 41 for problems larger than 30MiB. Hence writing code specifically for component grids seems worthwhile when the combina-
tion technique should be established for time dependent PDEs. Furthermore, hierarchization according to the unidirectional principle has bounded operational intensity for large problems. This result is not limited to component grids but holds for any (sparse) grid. Hence, to further improve the operational intensity the unidirectional principle needs to be avoided and algorithms cannot be restricted to work on 1-dimensional subproblems only.

The unidirectional hierarchization algorithm is optimized in the following way: first, the regular structure of the component grids enables us to navigate on the data layout, in particular to calculate the hierarchical predecessors, on the fly using integer strides for all dimensions. For regular or adaptive sparse grids the navigation on the data layouts is more involved, see Section 4.2. In particular, there is no need to calculate the level-index vector of a grid point. Second, we observe that poles for dimension \( r \geq 2 \) are perfect to be merged to large basic blocks. The grid points with same relative position in a set of consecutive poles, say the first point of each pole, are in contiguous memory. The same holds for their hierarchical predecessors. Hence, instead of looping over the poles one by one, we update grid points with the same relative position for a whole basic block of poles. This implies that the presented algorithm works orthogonal to the poles for \( r \geq 2 \) and not along the poles as the common unidirectional hierarchization algorithm does. Third, the grid is padded, i.e. unused grid points are added, such that the length of the poles in the first dimension is a multiple of the cache line size. This ensures, for all dimensions \( r \), that different cache lines either contain grid points from identical or disjoint sets of basic blocks of poles. Hence, basic blocks of poles can be handled independently of each other and are therefore ideal for parallelization. Within the basic blocks, the contiguous access to the grid points is, in addition, ideal for vectorization, i.e. the use of vector registers and instructions. Furthermore, if the size of the basic blocks is also chosen as a multiple of the cache line size, the padding ensures that only the faster, aligned versions of the vector instructions need to be used.

The unidirectional hierarchization algorithm of this chapter is implemented for component grids without boundary. This has solely historic reasons and it would be simple to modify the unidirectional hierarchization algorithm such that it can handle component grids with boundary points. As no boundary points are included, the third loop of Algorithm 5.1 only needs to go down to 2 instead of 1 as discussed in Section 4.8.1. Recall that the hierarchization algorithm for the piecewise-linear basis functions depicted in Figure 4.1 is optimized. Hierarchization for other basis functions is very similar but can require more floating point operations.

The rest of the chapter is organized as follows: the next section derives bounds for the runtime, the operational intensity and the flop
count of the unidirectional hierarchization algorithm. Section 5.3 describes the implementation of the unidirectional hierarchization algorithm \textit{combiHier}. Section 5.4 presents and discusses the experiments and Section 5.5 concludes the chapter.

**Research Contributions.** The results presented in this chapter have been published as \cite{Hup14b, Hup13} and this chapter contains large text parts of these publications.

### 5.2 Bounds for Unidirectional Hierarchization

This section derives a bandwidth bound and an operational intensity bound for unidirectional hierarchization assuming piecewise linear basis functions. It also analyzes the number of floating point operations performed by Algorithm 4.1.

#### 5.2.1 A Bandwidth Bound for the Unidirectional Principle

The unidirectional principle sweeps \( d \) times over the data. In each sweep the data is read, updated and written back. When the input is significantly larger than the cache, no intermediate results are reused between sweeps. (In theory, data up to cache size could be reused. This becomes irrelevant as the data is significantly larger than the cache.) Therefore, given the memory bandwidth of the processor, a lower bound for the runtime of any hierarchization algorithm implementing the unidirectional principle working on input data significantly larger than the cache is

\[
\frac{2d \cdot \text{(number of grid points)} \cdot \text{(size of datatype)}}{\text{bandwidth}}. \tag{5.1}
\]

#### 5.2.2 An Operational Intensity Bound for Unidirectional Hierarchization

In each sweep of the unidirectional principle at most 4 floating point operations are performed by Algorithm 4.1 to update one grid point. Given the assumption that no reuse is performed between the iterations of the unidirectional principle, at least the grid point itself must be read from main memory and its updated value written back. Hence the operational intensity of unidirectional hierarchization is upper bounded by

\[
\frac{4 \text{ flops}}{2 \cdot \text{(size of datatype)}}. \tag{5.2}
\]

In case of doubles the operational intensity is therefore bounded by

\[
0.25 \frac{\text{flops}}{\text{byte}}.
\]
To calculate the hierarchical surplus for basis functions different from the piecewise linear ones discussed here, not 4 but a different number of flops per grid point would be required in each sweep of the unidirectional principle. Hence, while the general approach stays valid, the value of the operational intensity bound Equation 5.2 would change if different basis functions would be employed.

### 5.2.3 Flop Count

To hierarchize a d-dimensional component grid of level \((\ell_1, \ldots, \ell_d)\) Algorithm 4.1 performs

\[
F(d, \ell) = 2 \cdot \sum_{r=1}^{d} \left( 2^{\ell_r+1} - 2 \cdot \ell_r - 2 \right) \cdot \prod_{s=1, s \neq r}^{d} \left( 2^{\ell_s} - 1 \right)
\]

(5.3)

flops. Observe that the flop count of Algorithm 4.1 could easily be reduced by roughly \(\frac{1}{4}\), if desired. Whenever both hierarchical predecessors exist, their values could be added first, multiplied by \(-0.5\) and then added to the value of \(x_g\). As the experiments are going to show that the hierarchization according to the unidirectional principle is memory bound and not compute bound, no experiments have been conducted with a reduced flop count version of the code. Also, reducing the flop count would reduce the operational intensity bound to \(\frac{3}{16} \text{ flops/byte} = 0.1875 \text{ flops/byte}\).

### 5.3 Implementation of the Unidirectional Hierarchization Algorithm

This section discusses the implementation of Algorithm 4.1. First, we specify how Algorithm 4.1 navigates on the data layout using the solely integer strides. Then optimizations of the algorithm are discussed.

The implementation of the unidirectional hierarchization algorithm for component grids discussed in this chapter works in-place as Algorithm 4.1. The input and output layout is standard row-major order as usual for full grids.

#### 5.3.1 Basic Navigation on the Data Layout

Algorithm 4.1 does not specify how to calculate the start of the next pole, how to loop over a certain level of a pole and how to compute the hierarchical predecessors. For adaptive or regular sparse grids, a lot of effort has been spent to derive efficient ways to navigate on sparse grids. As we are considering the anisotropic but full component grids, this navigation can be performed on the fly using integer strides. Integer
strides allow an efficient implementation and also reduce the memory overhead to a minimum.

For dimension $r$, there are $\prod_{1 \leq s \leq d, s \neq r} (2^{\ell_s} - 1)$ poles. The lexicographic first point of the $p$th pole in dimension $r$ is given by

$$\left\lfloor \frac{p}{\text{stride}_r} \right\rfloor \cdot \text{jump}_r + (p \mod \text{stride}_r) \quad \text{for}$$

$$\text{stride}_r = \prod_{s=1}^{r-1} (2^{\ell_s} - 1) \quad \text{and} \quad \text{jump}_r = \prod_{s=1}^{r} (2^{\ell_s} - 1).$$

Hence it is left to navigate within a pole of dimension $r$. A pole in dimension $r$ consists of $2^{\ell_r} - 1$ points in total or $2^{k_r-1}$ points for level $k_r \in \{1, \ldots, \ell_r\}$. The lexicographic first point of pole $P$ on level $k_r$ is at offset

$$2^{\ell_r-k_r} \cdot \text{stride}_r$$

from the lexicographic first point of the pole. Within level $k_r$, the points are at stride

$$2^{\ell_r-k_r+1} \cdot \text{stride}_r$$

If present, the hierarchical predecessors of a point are at stride

$$\pm 2^{\ell_r-k_r} \cdot \text{stride}_r$$

from the point itself.

The second loop of Algorithm 4.1 is parallelized using OpenMP as all poles can be handled independently of each other. As hierarchization of all poles requires the same number of floating point operations static scheduling with maximum chunk size was used. The unoptimized version of combiHier implements Algorithm 4.1 using these specifications looping over the poles in their natural order.

5.3.2 Optimizing the Unidirectional Hierarchization Algorithm

The unoptimized code has been optimized using basic block optimizations and manual vectorization. Hierarchizing in the first dimension has a special character and is not optimized further.

For work dimension $r \geq 2$ consider the arrangement of the poles as depicted in Algorithm 5.1 for $d = 2$ and $r = 2$. The poles (orange boxes) are orthogonal to the data layout and thus the data layout is perfect for block optimizations and vectorization (vector registers dashed). Loading a point of a pole automatically loads the whole cache line and page of that point. Both, cache line and page, span over different poles as we are working in row major layout. The relative position within its pole,
Algorithm 5.1 Basic block optimized unidirectional hierarchization algorithm for a $d$-dimensional component grid $C_\ell$ of level vector $(\ell_1, \ell_2, \ldots, \ell_d)$, using basic blocks of size $\text{block}$.

```
// Treat the first dimension as before
Hierarchize dimension 1 as in Algorithm 4.1

// Basic block optimizations for $r \geq 2$
for $r \leftarrow 2$ to $d$ do
  for each basic block $BL$ of $\text{block}$ poles in dim $r$ do
    for $k_r \leftarrow \ell_r$ down to 2 do
      for all indices $g$ such that $x_g$ is on level $k_r$ of
        the first pole of the basic block $BL$ do
        // Execute once for each pole in the basic block $BL$
        for $b \leftarrow 1$ to $\text{block}$ do
          $x_{g+b} = x_{g+b} - 0.5 * x_{\text{leftPredecessor}}(g+b, r)$
          $x_{g+b} = x_{g+b} - 0.5 * x_{\text{rightPredecessor}}(g+b, r)$
        end for
      end for
    end for
  end for
end for
```

however, is the same for all points of the same cache line and page (assuming the first dimension is refined sufficiently) as we are working on the anisotropic but full component grids. In particular,

$$x_{\text{rightPredecessor}}(g+b, r) = x_{\text{rightPredecessor}}(g, r)+b \cdot$$

The same holds for the left predecessor. Hence, the hierarchical predecessors of points that are contiguous in memory are contiguous in memory as well. This observation can be exploited to increase memory reuse. Instead of handling the poles after each other, basic blocks of $\text{block}$ poles are created. Within the first pole of a basic block we still work level by level bottom up and within a level position by position. The innermost loop, however, iterates over all poles of that basic block as different poles are at unit stride. Hence we finish the update of those contiguous points for several poles before we move on to the next grid point of the first pole of that basic block. This results in Algorithm 5.1. StructuredSG [BJPM12] employs a similar strategy to improve locality but can only treat hierarchical increment spaces, which are of much smaller size, as basic blocks. Instead of parallelizing the loop over the poles the loop over the basic blocks is parallelized for the optimized version.

The sequence of computations and the data layout are now ideal for vectorization. The scalar operations can simply be replaced by their corresponding vector operations.
Basic block optimizations as well as vectorization are simplified if the grid is padded such that the number of grid points in the first dimension is a multiple of the block size \( \text{block} \) and the size of the vector registers. This padding aligns all points of the first pole of a basic block to a multiple of the vector register size and hence enables us to use aligned loads and stores.

### 5.4 Experimental Results and Discussion

This section first describes the experimental setup. Then the experiments examining strong scaling, the size of the basic blocks, grids of different sizes and anisotropic grids are presented and a benchmark is performed. The implementation of `combiHier` is publicly available [Hup14a] and was verified for correctness using \( \text{SG}^{++} \) [Pfl10].

#### 5.4.1 Experimental Setup

All experiments were performed on an Intel Xeon E3-1240 running at 3.4GHz under Fedora 19. TurboBoost was disabled. The machine consists of 4 cores and hyperthreading was disabled. The 4 cores share 8MiB L3 cache and 32GiB main memory. As compiler `icc` version 14.0.1 with the flags `-std=c++0x -openmp -xHost -O3` was used. `gcc` version 4.8.2 was installed for library support and `OpenMP` version 3.1 employed.

All experiments were performed in double precision. For vectorization the 4-way AVX registers were used. The largest data sets examined for each dimension were roughly 8GiB \((\|\ell\|_1 = 30)\). If the level sum decreases by one, the size of the data set halves approximately. For \( d = 4 \) we had to limit the experiments to \( \ell_r = 7 \) \((\|\ell\|_1 = 28)\) resulting in data sets of roughly 2GiB. To choose the block size as a power of 2, the first dimension was padded with 1 grid point such that there were \(2^\ell_1\) grid points in the first dimension. The maximum refinement level as well as the standard size used for the basic blocks are summarized in Table 5.1.

The bandwidth was measured with the stream benchmark [McC95] and peaked at 21.1 GiB/s when at least 2 cores were used. The unpadded component grid consists of \(\prod_{r=1}^d (2^{\ell_r} - 1)\) grid points. To cal-
To analyze the performance of the implementation the Roofline Model [WWP09] is employed. To generate the roofline plots and to measure the data transfer between main memory and cache, *perfPlot* [OSC+14] is used. In particular, *perfPlot* guarantees cold cache measurements. For small basic blocks we observed that measurements from the flop counters differed by up to a factor of 2 from the value given by Equation 5.3. As *perfPlot* measures issued but not executed or retired flops we attribute this to wrong branching for small basic blocks. (For basic blocks of size 4 the loop body needs only to be executed once.) Hence the measured flops were replaced by the value calculated with Equation 5.3. As a consequence the performance numbers in the roofline plots are indirectly proportional to runtime.

For all other measurements the wall clock time was stopped using the *system_clock* of the C++ *chrono* library and warm cache measurements were performed. These experiments were repeated 10 times and the average results are shown. When error bars are plotted, these report the minimum and maximum over the 10 runs.

### 5.4.2 Experimental Results

**Scaling:** The optimized code is already bandwidth bound. The unoptimized code scales strongly. Figure 5.1 shows the wall clock time of the unoptimized and optimized code for different dimensions and maximum refinement level. While the unoptimized code scales almost perfectly for \( d = 3, 4, 5 \), the optimized code does not benefit from using more than 3 cores. The performance of the optimized code is already very close to the bandwidth bound and the overhead created by thread synchronization slows the algorithm down when 4 cores are used. The runtimes of the optimized code for 3 and 4 cores as well as the bandwidth bound of Equation 5.1 and the ratio \( \frac{\text{runtime}}{\text{bandwidth bound}} \) are summarized in Table 5.2. This table shows that the optimized version is within a factor of 1.5 of the bandwidth bound for all dimensions and respective maximum refinement level. Furthermore, this factor decreases as the dimension increases as hierarchizing the first dimension is not optimized. As the optimized version performs best when 3 cores are selected all upcoming measurements are performed using 3 cores and the optimized code.

**Basic block optimizations:** Basic blocks should be large. Figure 5.2 shows performance for grids of maximum refinement level when the size of the basic blocks is varied. If the first dimension is only moderately refined, the maximum block size should be chosen \( (d = 4, 5) \). If the first dimension is refined very finely, very large basic blocks may
5.4 Experimental Results and Discussion

Scaling of the Optimized and Unoptimized Code:
Optimized Code is Bandwidth Bound - Unoptimized Code Scales Strongly

![Graph showing runtime comparison]

Figure 5.1: Strong scaling of the unoptimized and optimized code for different dimensions. Perfect scaling for the unoptimized code is indicated by dashed lines.

<table>
<thead>
<tr>
<th>Grid Size ((r \in {1, \ldots, d}))</th>
<th>Bandwidth Lower Bound (Equation 5.1)</th>
<th>Number of Processors</th>
<th>Runtime [s]</th>
<th>Bandwidth Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d = 2, \ell_r = 15)</td>
<td>1.52 s</td>
<td>3</td>
<td>2.15 s</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.12 s</td>
<td>1.39</td>
</tr>
<tr>
<td>(d = 3, \ell_r = 10)</td>
<td>2.27 s</td>
<td>3</td>
<td>3.33 s</td>
<td>1.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3.79 s</td>
<td>1.66</td>
</tr>
<tr>
<td>(d = 4, \ell_r = 7)</td>
<td>0.73 s</td>
<td>3</td>
<td>0.84 s</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0.86 s</td>
<td>1.18</td>
</tr>
<tr>
<td>(d = 5, \ell_r = 6)</td>
<td>3.50 s</td>
<td>3</td>
<td>4.19 s</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>4.85 s</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Table 5.2: Comparing the runtime of the optimized code for 3 & 4 cores to the bandwidth bound.
decrease performance \((d = 2, 3)\). The best block size improves performance between 1.5 and 3 for all dimensions compared to basic blocks of size 4. For \(d = 2, 3\) increasing the size of the basic blocks also increases operational intensity as larger basic blocks increase temporal locality. For \(d = 4, 5\) this effect does not appear as the poles are shorter and the cache can hold sufficient entries such that data is not evicted while finishing a basic block. Also, for large basic blocks and all dimensions, the algorithm performs close to the bandwidth bound of Equation 5.1 and the operational intensity bound of Equation 5.2.

**Varying the Grid Size: Large Problems Are Bandwidth Bound.** Figure 5.3 shows the performance for different dimensions and refinement levels. The lower plot enlarges the critical region of the upper plot. The plots show that the bound on operational intensity limits the performance achievable for large data sets to less than \(1/16\) of the available AVX peak performance. For small levels, the data sets are small enough to fit into cache and do not need to be written back to main memory between the loops of the unidirectional principle. Hence, in these circumstances the code is able to beat the operational intensity upper bound of Equation 5.2. As soon as the data sets do no longer fit into cache \((d = 2 & \ell_r = 11, d = 3 & \ell_r = 7, d = 4 & \ell_r = 6, d = 5 & \ell_r = 5)\) the operational intensity abruptly drops below the operational intensity bound Equation 5.2 of \(0.25 \text{flops/cycle}\). Again, one can observe that the performance is close to the bandwidth bound Equation 5.1 and to the operational intensity bound Equation 5.2 for large grids.
Figure 5.3: Increasing the level for different dimensions. The lower plot enlarges the critical region.
Anisotropic Grids for Dim. 6: Isotropic Grids can Serve as Upper Bound

Isotropic Grids: Baseline and Upper Bound for Anisotropic Grids. As the combination technique also requires anisotropic grids, we compare the time spent hierarchizing an isotropic grid with times spent when a highly anisotropic grid of the same refinement level is hierarchized. The experiments are conducted for \( d = 6 \) and the level sum is held fix at \( \|\ell\|_1 = 30 \). The anisotropic grid has refinement level \( \ell_r = 5 \) for all dimensions \( r \). For the anisotropic grids, one dimension is refined to level 15 while all other dimensions have refinement level 3. For \( \ell_1 \in \{3, 5\} \), block was chosen as \( 2^{\ell_1} \) and for \( \ell_1 = 15 \) block was set to \( 2^{14} \). The bandwidth bound differs for the isotropic and anisotropic grids as the number of grid points differs.

Figure 5.4 shows that hierarchizing the isotropic grid takes longer than hierarchizing any of the anisotropic grids. Hence the experiments conducted for the isotropic grids can serve as an upper bound for the runtime needed to hierarchize an anisotropic grid of the same refinement level and dimension. In addition, hierarchizing the anisotropic grids is faster if the refined dimension is low. In particular, if the first dimension is refined, the runtime is very close to the bandwidth bound as a large refinement level in the first dimension enables large basic blocks.

Benchmark: By exploiting the regular structure of component grids we can outperform generic software by about 1 order of magnitude for large grids. To show the competitiveness of \textit{combiHier}it is benchmarked against \textit{StructuredSG}, the currently fastest hierarchization code. As \textit{StructuredSG} is not limited to component grids it cannot exploit their full structure. While \textit{combiHier} uses a full grid layout, \textit{StructuredSG} splits a component grid according to its hierarchical increment spaces. Whether this increment space splitting...
Hierarchization describes the base change from the full grid basis to the hierarchical basis which allows sparse grids to lessen the curse of dimensionality. Besides being an important sparse grid algorithm for that reason, the hierarchization algorithm is also prototypical for sparse grid algorithms. This means that the optimizations performed for the hierarchization algorithm are also likely to be beneficial for other sparse grid algorithms. In addition, the hierarchization algorithm and its inverse
transformation dehierarchization are necessary as intermediate steps when the sparse grid combination technique employs algorithms that work in both the full grid basis and the hierarchical basis.

This chapter has discussed an implementation of the hierarchization algorithm that is tuned specifically for the component grids of the combination technique. Experiments have demonstrated that the presented implementation `combiHier` achieves close to maximum operational intensity and comes within less than a factor of 1.5 from the bandwidth bound imposed by the unidirectional principle (see Equation 5.1). Furthermore, `combiHier` is able to outperform generic software by a factor between 5.8 and 41 for problems larger than 30MiB. Hence, writing code specifically for component grids seems worthwhile.

As hierarchization is prototypical for sparse grid algorithms, future work can use the presented optimizations to speed up other sparse grid algorithms. As the optimizations presented in this chapter exploit the structure of the component grids, they are mainly applicable to algorithms that are relevant for the combination technique and its component grids. One such algorithm is the dehierarchization algorithm which describes the inverse change of basis, i.e., from the hierarchical basis to the full grid basis. Implementing a dehierarchization algorithm that is tuned for component grids should be straightforward using the discussed optimizations.

This chapter has optimized the unidirectional hierarchization algorithm for component grids. As a result, this algorithm cannot be applied to regular or adaptive sparse grids which have more complicated structure. Also, no alternatives to the d global sweeps of the unidirectional principle have been explored. In consequence, the unidirectional lower bound of Equation 5.1 limits the performance of this implementation as soon as the component grids do no longer fit into cache. It was, however, shown that the presented implementation comes close the lower bound given by the unidirectional principle. To beat this bound, the d global phases of the unidirectional principle need to be avoided. Chapter 6 develops such an approach using a divide and conquer strategy.
6.1 Introduction

So far, all hierarchization algorithms implement the unidirectional principle. Furthermore, in Chapter 5 we have seen a memory efficient implementation of the unidirectional hierarchization algorithm for component grids which is within a factor of 1.5 of the optimal runtime for unidirectional algorithms. While the unidirectional principle performs few floating point operations as discussed in Section 4.9, any unidirectional algorithm is inherently memory inefficient. The characteristic of the unidirectional principle are the \( d \) global sweeps over the data which results in at least

\[
d \cdot \frac{|C_\ell|}{B} - (d - 1) \cdot \frac{M}{B} = d \cdot \frac{1}{B} \cdot \left( |C_\ell| - \frac{d - 1}{d} \cdot M \right).
\]

cache misses as it was stated in Equation 4.2. If the grid is significantly larger than the internal memory, i.e. \( M = o(|C_\ell|) \), the reuse between sweeps is negligible. Hence, for large grids, unidirectional algorithms are, by design, not cache-efficient as they access each grid point \( d \) times. Any significant further improvements have to avoid the global unidirectional principle.

This immediately raises the question: can we design a hierarchization algorithm that beats the lower bound for the memory accesses of the unidirectional hierarchization algorithm? Can we reduce the number of accesses to each grid point to less than \( d \) by avoiding the \( d \) global sweeps of the unidirectional principle?

This chapter derives a divide and conquer hierarchization algorithm for isotropic component grids \( C_\ell \) that reduces the leading term of the number of cache misses by a factor of \( d \) compared to the unidirectional lower bound given a tall cache of size \( M = \omega (B^d) \). The complexity of the algorithm in the cache-oblivious model [Pro99, FLPR99] is

\[
|C_\ell| \cdot \left( \frac{1}{B} + o \left( \frac{1}{\sqrt{M}} \right) \right)
\]

and the tall cache assumption guarantees that the second term is indeed of lower order. The constant at the leading term of memory accesses is optimal, as any hierarchization algorithm has to read the input causing \(|C_\ell|/B\) cache misses. The divide and conquer algorithm achieves this performance by avoiding the unidirectional principle globally. Instead, it applies the unidirectional principle recursively to smaller subproblems.
using a divide and conquer strategy. The algorithm works on either the standard row- or column-major order data layout.

We further complement the upper bound with a lower bound of

\[ |C_\ell| \cdot \left( \frac{1}{B} + \Omega \left( \frac{1}{B} \cdot \frac{1}{d - \sqrt{M}} \right) \right) \]

assuming \(|C_\ell| \in \omega(M^{d+1})\). This assumption guarantees that the input is significantly larger than the cache and is slightly stronger than the usual assumption \(M \in o(|C_\ell|)\). This is the first non-trivial lower bound, i.e., not the scanning bound, proven for the hierarchization task. For \(B = 1\), a minor modification of the algorithm shows that its complexity is \(|C_\ell| \cdot \left( 1 + \Omega \left( \frac{1}{d - \sqrt{M}} \right) \right)\) and hence the lower and upper bound match for \(B = 1\). The lower bound is limited to algorithms that can be expressed as linear arithmetic circuits, i.e. algorithms that can compute linear combinations but not arbitrary algebraic or transcendental functions of the input. This limitation enables us to derive an isoperimetric or rank argument and which is applied after the algorithm has been split into rounds. This technique is very similar to that employed for stencil computations in Section 3.5 and was originally developed by Hong and Kung [HK81]. While we analyze the algorithm in the cache-oblivious model, we prove the lower bound in the cache-aware external memory [HK81, AV88] model as this only strengthens the lower bound. Also, the tall cache assumption is not needed for the lower bound.

The divide and conquer algorithm presented in this chapter is an algorithmic idea and still has to be implemented to show its relevance for practical purposes. As we analyze the algorithm in the theoretical cache-oblivious model, the analysis is limited to the effects captured in this model (see Section 2.1.4). The divide and conquer idea can also be used to design a cache-aware hierarchization algorithm (see Section 6.4). A cache-aware version would be simpler to implement and, presumably, further increase the performance of the algorithm. Also, the tall cache assumption \(M = \omega(B^d)\) is strong. The algorithmic idea of merging several phases of the unidirectional principle, however, can also be used to reduce the number of cache misses for smaller caches (see Section 6.4 once more).

This chapter discusses the divide conquer algorithm for isotropic component grids \(C_\ell\), i.e. component grids that are refined equally in all dimensions. Formally, \(\ell_r = c \forall r \in \{1, \ldots, d\}\). While this assumption is not essential for the lower bounds, it simplifies the design of the algorithm. If the component grid is isotropic, all of the subgrids of the recursion have the same size and are isotropic as well. Furthermore, we can use any of the components of \(\ell\), e.g. \(\ell_1\), when we scale the grid. See Section 6.4 for a generalization of the divide and conquer approach
to anisotropic component grids and regular as well as adaptive sparse grids.

Furthermore, we consider component grids with boundary of type 1, i.e., the boundary is refined as the interior of the grid. The boundary grid points ensure that all interior grid points have exactly two hierarchical predecessors per dimension. This assumption is used in the proof of the lower bound in Lemma 6.11, when we choose the hierarchical predecessors for the direct poles and know that such a hierarchical predecessor always exists. As basis functions, the standard basic piecewise-linear basis functions are employed. Hence, Algorithm 4.1 and Algorithm 4.2 hierarchize the grid.

Refer to Chapter 4 for notation and background regarding sparse grids. To ease readability and in agreement with common usage in the sparse grid literature, this chapter generally assumes for the $O$-notation that the dimension $d$ is constant. See Section 4.1 for the discussion why the dimension $d$ is treated as a constant in the sparse grid context. For completeness, we also state the complexity of the lower and upper bound at the end of the relevant section including the constant $d$. The tall cache assumption is $M \in \omega \left( (d^2 \cdot B)^d \right)$ when the constant $d$ is included. The upper bound also requires the technical assumption $B \leq \frac{1}{2d} \cdot \sqrt[4]{M/6}$ which is a version of the tall cache assumption with fixed constants. In this chapter, $r$ or $s$ are used to index $d$-dimensional vectors. Therefore, if not further specified, it is assumed that $r \in \{1, \ldots, d\}$ and $s \in \{1, \ldots, d\}$.

The rest of this chapter is organized as follows: Section 6.2 first gives an overview of the upper bound, then states the divide and conquer algorithm and subsequently analyzes it in the cache-oblivious model. Subsequently, the non-trivial lower bound for the number of non-compulsory I/Os is discussed in Section 6.3 which first gives an overview of the lower bound before deriving it in detail. The chapter then discusses generalizations of the divide and conquer algorithm and concludes subsequently.

research contributions. The results presented in this chapter are joint work with Riko Jacob and are so far unpublished [HJ14a]. This chapter contains large text parts of the current draft of this manuscript.

6.2 Upper Bound

6.2.1 Overview of the Upper Bound

The upper bounds builds upon a tall cache assumption of $M \in \omega (B^d)$. In addition, it requires a version of the tall cache assumption with fixed constants, namely $B \leq \frac{1}{2d} \cdot \sqrt[4]{M/6}$. We further assume that the input grid $C_\ell$ is significantly larger than the cache, i.e. $M = o(|C_\ell|)$. Other-
wise, constant fractions of the input would fit into cache and the problem changes significantly. The problem would also become irrelevant in practice as it can be solved by filling the whole cache a constant number of times (take $d-1$ dimensional slices as subproblems). Hence, the runtime would be negligible. The algorithms are designed for a standard row- or column major order.

The upper bound introduces a new algorithmic idea for sparse grids using the divide and conquer technique. The goal is to avoid the unidirectional principle on a global scale to avoid its $d$ passes over the data, but apply it recursively to smaller subproblems using the unidirectional hierarchization algorithm for sets (Algorithm 4.2). Recall that we need at least access to all direct hierarchical predecessors of $S$ when we want to hierarchize $S$ with Algorithm 4.2. Hence, we want to choose those sets $S$ as subproblems that have few direct hierarchical predecessors outside of $S$. Subgrids in the hierarchical sense are a good choice. This does not mean the hierarchical increment grids or spaces, but those geometric subgrids that form the orthants of the current problem. These orthant subgrids can be described as the left, right or all descendants of few grid points. We choose the subproblems $S$ to be the interior of the orthant subgrids. All hierarchical predecessors of $S$ with respect to the $3^d$-stencil are either in $S$ or in the boundary of the orthant subgrid itself (see Figure 6.1 for a 1-dimensional example and refer to Lemma 6.1 for the proof of this claim). This is sufficient to hierarchize the interior of the subgrid correctly with Algorithm 4.2 (compare to Theorem 6.3). As the number of (interior) grid points of a subgrid is exponential in $d$ while the boundary of a subgrid is exponential in $d-1$, this approach is going to be efficient.

The boundary vertices, however, cannot be hierarchized correctly although we alter their values during the execution of Algorithm 4.2. To hierarchize the boundary vertices correctly at a coarser level of the recur-
sion, we make a back-up copy of their original values. This enables us to calculate the correct hierarchical surplus when the former boundary vertex is now in the interior of a coarser subgrid. Performing recomputations is a common tool for designing memory efficient algorithms. The recomputations ensure that we can use Algorithm 4.2 as an “as is” component for the divide and conquer algorithm.

To back up and buffer the original values of the boundary of the current subproblem we need $4d \cdot \left| C_\ell \right|^{d-1}$ additional memory. Therefore, the divide and conquer algorithm cannot work completely in-place as the unidirectional hierarchization algorithm does.

The divide and conquer approach is applied to the hierarchization algorithm for isotropic component grids, i.e. we assume that the component grid is refined equally in all dimensions. This assumption facilitates the analysis of the divide and conquer algorithm but does not limit its applicability (see Section 6.4). Furthermore, it is assumed that the component grid is stored in standard row- or column-major order. Hence, the subproblems of the recursion are row-wise respectively column-wise in contiguous memory as they are subgrids. This is exploited in the proof of Lemma 6.6 for the row-major order and works for the column-major order in the same way. This means that the data fits into internal memory and hence a single sweep over the data suffices to hierarchize the interior of the current subproblem when the number of grid points of the subproblem is roughly $M$ (and not $M/B$). As we need to access the whole subgrid while we can only hierarchize its interior implies that we need to read more data than we can hierarchize. This overhead leads to the term $O\left(\left| C_\ell \right| / \sqrt{M} \right)$ which is of lower order given a tall cache of $M \in \omega\left(B^d\right)$. Few vertices, the boundary vertices of the subproblems of size roughly $M$, are unhierarchized after we have finished all those subproblems. These are so few vertices that they can be hierarchized in a brute-force manner using at most $O\left(\left| C_\ell \right| / \sqrt{M} \right)$ cache misses. I.e., no particular attention needs to be paid to perform these coarse levels of the recursion in a memory efficient manner, given the tall cache assumption.

The divide and conquer algorithm presented in this chapter is an algorithmic idea. In particular, as we analyze the algorithm in the theoretical cache-oblivious model, the analysis is limited to the effects captured in this model (see Section 2.1.4). On a real machine, many more effects would need to be taken into account. For example, the cache is assumed to be fully associativity and the ideal cache of the cache-oblivious models assumes that the best cache replacement strategy is used.

For the presented algorithm, the cache replacement strategy is only relevant for small subproblems and hence in the proof of Lemma 6.6. For the coarse levels of the recursion, Lemma 6.5, we only assume that 3 blocks fit into cache at once and hence the cache replacement strategy can be neglected. The cache replacement strategy that we analyze reserves roughly half of the memory to back up the original values for the
boundary vertices. These memory blocks are always kept in memory. For the other half of the cache a least recently used (LRU) strategy is used. In Lemma 6.6 the subproblems are chosen such that they fit into this second half of the cache. Hence, while working on it, we never evict cache lines of the current subproblem. If the whole cache would use a LRU strategy, we could use dummy accesses to the first half of the cache to simulate the described cache replacement strategy.

Furthermore, the coarse levels of the recursion are treated in a brute-force manner in the algorithmic design as well as in the analysis. As a result, the “lower order” term dominates if the tall cache assumption is not met. In this case, the complexity of the divide and conquer algorithm is worse than that of the unidirectional algorithm.

6.2.2 Derivation of the Upper Bound

The algorithm is going to work recursively on subgrids and hence we need notation to describe the subgrids of a component grid \( C_\ell \). A subgrid \( G(x, k, \ell) \subset C_\ell \) is described by its offset \( x \), the refinement level \( k \) of the subgrid and the refinement level \( \ell \) of the super grid \( C_\ell \). The subgrid \( G(x, k, \ell) \) has its “first” grid point at the offset \( x \) and then extends in the positive unit directions. It has mesh width \( 2^{-\ell_r} \) in dimension \( r \) and \( 2^{k_r} \) grid points in that dimension.

\[
G(x, k, \ell) := \{ y \in [0, 1]^d : \forall r : \exists i_r \in \{0, 1, \ldots, 2^{k_r}\} : y_r = x_r + i_r \cdot 2^{-\ell_r} \}.
\]

In particular, \( G(0, 0, \ell) = C_\ell \).

We are interested in the subgrids that decompose the grid \( G(x, k, \ell) \) into the \( 2^d \) subgrids of its orthants. The boundaries of these orthant subgrids overlap. These subgrids are hierarchical in the sense that, for an interior grid point, all hierarchical predecessors given by the \( 3^d \)-point stencil are in the subgrid itself (see Lemma 6.1). Hence, all interior points of the subgrid are hierarchized correctly when we apply the unidirectional hierarchization algorithm to the subgrid (see Theorem 6.3).

These orthant subgrids are refined once less in each dimension than their supergrid. Hence, they contain roughly half of the grid points of \( G(x, k, \ell) \) per dimension. For \( \delta \in \{0, 1\}^d \), the orthant subgrids of \( G(x, k, \ell) \) are

\[
\text{subGrids} \left( G(x, k, \ell) \right) := \{ G(x', k', \ell) \subset G(x, k, \ell) : k'_r = (k_r - 1) \ \forall r \ \text{and} \ \ x' = x + \sum_{r=1}^d \delta_r \cdot 2^{-(\ell_r - k'_r)} \cdot e_r \ \text{with} \ \delta_r \in \{0,1\} \}.
\]

As we assume that the initial component grid \( C_\ell \) is refined equally in all dimension, all subgrids are so also.
Figure 6.2: Left: a 2-dimensional grid and its recursive splitting into its $2^d$ subgrids. Middle: base case of the recursion, i.e. $k = 1$. Right: recursive progress of the divide and conquer algorithm. The green hatched parts have been hierarchized correctly. Vertices that have been in cache but are not yet hierarchized are depicted in red. ($B = 1$ is assumed for this figure.)

The boundary $\square(G(x, k, \ell))$ and the interior $\upbrace(G(x, k, \ell))$ of the grid are defined in the canonical way, i.e. the boundary vertices are the outermost vertices of the grid in each dimension and the interior vertices are all vertices that are not in the boundary. The divisional surface $\boxtimes(G(x, k, \ell))$ are those interior vertices that split the grid into its orthants and hence subgrids. The divisional surface and boundary $\boxplus(G(x, k, \ell))$ is the (disjoint) union of divisional surface and the boundary. Formally:

$$\square(G(x, k, \ell)) := \left\{ y \in G(x, k, \ell) : \exists r : y_r = x_r + i_r \cdot 2^{-\ell_r} \text{ for } i_r \in \{0, 2^{k_r}\} \right\},$$

$$\upbrace(G(x, k, \ell)) := \left\{ y \in G(x, k, \ell) : \forall r : y_r = x_r + i_r \cdot 2^{-\ell_r} \text{ for } i_r \in \{1, \ldots, 2^{k_r} \} \right\},$$

$$\boxtimes(G(x, k, \ell)) := \left\{ y \in \upbrace(G(x, k, \ell)) : \exists r \in \{1, \ldots, d\} : y_r = x_r + i_r \cdot 2^{-\ell_r} \text{ for } i_r = 2^{k_r} - 1 \right\} \text{ and }$$

$$\boxplus(G(x, k, \ell)) := \square(G(x, k, \ell)) \cup \boxtimes(G(x, k, \ell)).$$

For $d = 2$, the left part of Figure 6.2, depicts the interior of the $2^d$ subgrids in green and numbers the subgrids from 1 to 4. The divisional surface is depicted in yellow (number 5) and the (global) boundary in red. The boundary of each of the 4 subgrids also contains a part of the divisional surface and the global boundary.

---

1 In 2d the divisional surface has the appearance of a cross.
The size of the boundary, the divisional surface and the divisional surface and boundary are bounded as follows:

\[
\Box(G(x, k, \ell)) \leq 2d \cdot \left(2^{k_1} + 1\right)^{d-1}
\]

\[
\check{\Box}(G(x, k, \ell)) \leq d \cdot \left(2^{k_1} + 1\right)^{d-1}
\]

\[
\Xi(G(x, k, \ell)) \leq 3d \cdot \left(2^{k_1} + 1\right)^{d-1}.
\]

The algorithm is going to update the grid points in-place but needs additional memory to store backup copies of the input values. This additional memory is \(Z_{k_1}\) \((1 \leq k_1 \leq \ell_1)\) and is used to backup the boundary of the current subproblem. Hence, we choose \(Z_{k_1}\) to be of size \(|Z_{k_1}| := \Box(G(0, k, k))\). Each \(Z_{k_1}\) is stored in its own contiguous block of memory. The size of the \(Z_{k_1}\) grows exponentially in \(k_1\). Therefore the total amount of memory is a geometric sum which is bounded by a constant times the base case. Thus, the total amount of additional memory is limited by

\[
\sum_{k_1=1}^{\ell_1} |Z_{k_1}| \leq 2 \cdot |Z_{\ell_1}| = 2 \cdot 2d \cdot \left(2^{\ell_1} + 1\right)^{d-1} = 4d \cdot |C_{\ell_1}|^{d-1}.
\]

Algorithm 6.1 states the divide and conquer hierarchization algorithm of this chapter. This algorithm works recursively and is cache-oblivious. The algorithm uses the recursive splitting of a grid into its subgrids and Algorithm 4.2 to hierarchize the divisional surface of a subgrid while accessing the divisional surface and boundary of the subgrid. On the right, Figure 6.2 depicts the recursive structure of Algorithm 6.1. Vertices that have been in cache but are not yet hierarchized are depicted in red and disclose the recursive pattern. These vertices are always boundary vertices of the respective subgrids. We first prove the correctness of Algorithm 6.1 and then analyze its complexity in the cache-oblivious model. For the correctness, we first show that all hierarchical predecessors with respect to the \(3d\)-stencil of vertices of the divisional surface are either in the divisional surface or the boundary (Lemma 6.2). This enables Algorithm 6.1 to hierarchize the divisional surface correctly if it has access to the divisional surface and boundary. In the following, we index a grid point \(y\) with its level-index vector \((\ell, i)\) when we want to stress the relation to its hierarchical predecessor. We also index the grid points as well as the level and the index vectors with the dimension \(r\) to denote their \(r\)th coordinate. The \(r\)th coordinate of the grid point \(y_{\ell, i}\) is denoted as \((y_{\ell, i})_r\).

**Lemma 6.1.** Let \(G(x, k, \ell)\) be any of the subgrids for which the function \(\text{hierarchizeRec}(G(x, k, \ell))\) is called in Algorithm 6.1. For any point \(y_{\ell, i}\) in the interior of the subgrid it holds that the hierarchical predecessors of \(y_{\ell, i}\) with
Algorithm 6.1 Divide and conquer hierarchization algorithm.

1: function hierarchizeRec(G(x_0, k, ℓ))
2:  if k_1 > 1 then // split G into 2^d subgrids of level k_1 - 1
3:     for G(x'_0, k', ℓ) ∈ subGrids(G(x_0, k, ℓ)) do
4:         hierarchizeRec(G(x'_0, k', ℓ))
5:  end for
6:  end if
7:  Copy □(G(x_0, k, ℓ)) from G(x_0, k, ℓ) to Z_{k_1},
8:  unidirHierarchize(⊞⊞⊞(G(x_0, k, ℓ))) // from Algorithm 4.2
9:  if k < ℓ then
10:     Copy □(G(x_0, k, ℓ)) from Z_{k_1} to G(x_0, k, ℓ)
11:  end if
12: end function

respect to the 3^d-stencil are in the subgrid itself. Formally, if (ℓ', i') denotes the reduced level of y_{ℓ,i},

\[ y_{ℓ,i} ∈ \bigoplus G(x, k, ℓ) \quad ⇒ \quad ∀ a ∈ \{-1, 0, 1\}^d : \quad y_{ℓ', (i' + a)} ∈ G(x, k, ℓ). \]

Proof. Choose any point y of the interior of the grid, i.e.

\[ y = y_{ℓ,i} ∈ \bigoplus G(x, k, ℓ). \]

Choose any dimension r. By definition,

\[ y_r = i_r · 2^{-ℓ_r} = x_r + j_r · 2^{-ℓ_r} \quad \text{for} \quad j_r ∈ \{1, \ldots, 2^{k_r} - 1\}. \]

To apply the 3^d-stencil and get the relevant hierarchical predecessors, we need the reduced level (ℓ', i') of y_{ℓ,i}. Therefore write j_r as

\[ j_r = \sum_{m=0}^{k_r-1} δ_m · 2^m \quad \text{for some} \quad δ_m ∈ \{0, 1\}. \]

In particular, δ_m = 1 for at least one m ∈ \{0, \ldots, k_r - 1\}. Denote by m_0 the smallest m such that δ_m = 1.

The function hierarchizeRec(G(x, k, ℓ)) is only called recursively for (orthant) subgrids of (orthant) subgrids. Hence, each coordinate of x is of the form

\[ x_r = \sum_{m=0}^{ℓ_r} δ_m · 2^{(ℓ_r - k_r)} \quad \text{for} \quad δ_m ∈ \{0, 1\}. \]
Therefore, the index vector of \( x \) is too large to influence whether \( y_{\ell,i} \) can be reduced. Formally,

\[
y_r = x_r + j_r \cdot 2^{-\ell_r} = \left( \sum_{m=0}^{\ell_r} \delta_m \cdot 2^m \right) \cdot 2^{-\ell_r} = \\
\left( \sum_{m=m_0}^{\ell_r} \delta_m \cdot 2^m \right) \cdot 2^{-\ell_r} = \left( \sum_{m=0}^{\ell_r-m_0} \delta_{m+m_0} \cdot 2^m \right) \cdot 2^{-\ell_r-m_0} =: i'_r.
\]

As the index \( i'_r \) is odd, \( (\ell_r - m_0, i'_r) \) is the reduced level-index vector in dimension \( r \).

With the reduced level-index vector, we can calculate the level-index vectors of the hierarchical predecessors in the \( 3^d \)-stencil. In each dimension \( r \), only grid points whose level-index vector is of the form \( (\ell_r - m_0, i'' + a) \) for \( a \in \{-1, 0, 1\} \) are in the \( 3^d \)-stencil of \( y_{\ell,i} \). If \( a_r = 0 \) then the index \( i'' \) does not change with respect to that dimension. If \( a_r = 1 \), then

\[
(y(\ell_r-m_0, i''+1))_r = x_r + \left( \sum_{m=m_0}^{k_r-1} \delta_m \cdot 2^m \right) \cdot 2^{-\ell_r} = \\
=x_r + \left( \sum_{m=m_0+1}^{k_r} \delta_m \cdot 2^m \right) \cdot 2^{-\ell_r} =: j''_r.
\]

Hence, \( j''_r \in \{0, \ldots, 2^{k_r}\} \).

Similarly, if \( a_r = -1 \),

\[
(y(\ell_r-m_0, i'-1))_r = x_r + \left( \sum_{m=m_0+1}^{k_r-1} \delta_m \cdot 2^m \right) \cdot 2^{-\ell_r} = \\
=x_r + \left( \sum_{m=m_0}^{k_r-1} \delta_m \cdot 2^m \right) \cdot 2^{-\ell_r} =: j''_r
\]

as \( \delta_{m_0} = 1 \). In particular, \( j''_r \in \{0, \ldots, 2^{k_r}\} \).

Hence, by definition of the subgrid, the hierarchical predecessors of the \( 3^d \)-stencil are within the subgrid \( G(x, k, \ell) \) itself.

\[\square\]

**Lemma 6.2.** Let \( G(x, k, \ell) \) be any of the subgrids for which the function \( \text{hierarchizeRec}(G(x, k, \ell)) \) is called in Algorithm 6.1. For any point \( y_{\ell,i} \) in the divisional surface of the subgrid it holds that the hierarchical predecessors of \( y_{\ell,i} \) with respect to the \( 3^d \)-stencil are either in the divisional surface or the
boundary of the subgrid itself. Formally, if \((\ell', i')\) denotes the reduced level of \(y_{\ell, i}\),

\[
y_{\ell, i} \in \Xi(G(x, k, \ell)) \implies \forall a \in (-1, 0, 1)^d : \ y_{\ell', (i'+a)} \in \Xi(G(x, k, \ell)) .
\]

**Proof.** Choose any point \(y\) in the divisional surface of the grid, i.e.

\[
y = y_{\ell, i} \in \Xi(G(x, k, \ell)) .
\]

By definition, the divisional surface is a subset of the interior and hence **Lemma 6.1** states that the relevant hierarchical predecessors are in the subgrid \(G(x, k, \ell)\). It is left to show that the hierarchical predecessors with respect to the \(3^d\)-stencil for vertices \(y_{\ell, i}\) of the divisional surface are in the boundary of the subgrid. Denote by \(r\) the dimension for which

\[
y_r = x_r + 2^{kr - 1} \cdot 2^{-\ell_r} .
\]

We determine the reduced level-index vector of \(y_{\ell, i}\) similarly to the proof of **Lemma 6.1**. Obviously, \(j_0 = 2^{kr - 1}\) and hence the reduced level-index vector in dimension \(r\) is \((\ell'_r, i'_r) := (\ell_r - (k_r - 1), 1)\).

Let us now calculate the \(r\).th coordinate of the level-index vector of an hierarchical predecessor of \(y_{\ell, i}\) with respect to the \(3^d\)-stencil. If \(a_r = 1\), the \(r\).th coordinate of the hierarchical predecessor is

\[
\left( y_{(\ell_r - (k_r - 1), i'+a)} \right)_r = x_r + 2^{kr} \cdot 2^{-\ell_r} .
\]

and hence \(y_{(\ell_r - (k_r - 1), 2)} \in \square(G(x, k, \ell))\) by definition.

If \(a_r = -1\), then

\[
\left( y_{(\ell_r - (k_r - 1), i'+a)} \right)_r = x_r + 0 \cdot 2^{-\ell_r} .
\]

and hence \(y_{(\ell_r - (k_r - 1), 0)} \in \square(G(x, k, \ell))\) by definition.

If \(a_r = 0\), the level-index vector of dimension \(r\) remains unchanged and the condition \(i_r = 2^{kr - 1}\) stays valid. As we know by **Lemma 6.1** that we stay within the subgrid, the only possibility for \(y_{\ell, i'+a}\) to be outside of the divisional surface is to be in the boundary of the subgrid.  

**Theorem 6.3.** The call hierarchizeRec\((G(0, \ell, \ell))\) (**Algorithm 6.1**) calculates the hierarchical surpluses of \(C_\ell = G(0, \ell, \ell)\) correctly.

**Proof.** We prove an invariant of **Algorithm 6.1**: Before Step 7 the values of \(\Xi(G(x, k, \ell))\) are the input values. After Step 11 the divisional surface \(\Xi(G(x, k, \ell))\) stores the hierarchical surpluses. The boundary stores the original values if \(k < \ell\) or the hierarchical surpluses if \(k = \ell\).

When we arrive at Step 7 for the first time the invariant holds. Hence, assume we are at Step 7 at any execution point of the algorithm and \(\Xi(G(x, k, \ell))\) stores the input values. We then backup \(\square(G(x, k, \ell))\) to \(Z_{k_1}\). Next we call the unidirHierarchize-routine (**Algorithm 4.2**) which
applies the standard unidirectional hierarchization algorithm to the set \( \mathcal{H}(G(x_k, \ell)) \). In particular by Lemma 6.2, for all vertices of the divisional surface all hierarchical predecessors of the \( 3^d \)-stencil are within \( \mathcal{H}(G(x_k, \ell)) \). This guarantees that the hierarchical surpluses of the vertices of the divisional surface can be computed correctly.

In detail: as we are working bottom up (by decreasing the level), the loop over the first dimension of Algorithm 4.2 calculates the hierarchical surpluses of all interior points of all poles of \( \mathcal{H}(G(x_k, \ell)) \) as we would call Algorithm 4.2 for the complete grid \( C_\ell \). The two outermost (boundary) points of each pole in the first dimension cannot be updated as their hierarchical predecessors are not part of \( \mathcal{H}(G(x_k, \ell)) \). Call this set of points which is not updated correctly with respect to the first dimension \( U_1 \). For the upcoming work dimensions \( s > 1 \), the points of \( U_1 \) are only part of poles that lie completely within the boundary. As we are only working within poles, a point in the boundary of a pole in the first dimension only contributes to points on the boundary of the grid in all upcoming dimensions. Hence values from \( U_1 \) are never input values to any points of \( \mathcal{H}(G(x_0, k, \ell)) \). The same argument holds for all other work dimension \( r \). The vertices which are not correctly updated after the first \( r \) loops of the unidirectional hierarchization algorithm are the \( 2 \cdot r \) faces of the boundary of the subgrid which correspond to the first \( r \) dimensions. As the unidirectional hierarchization algorithm proceeds, these incorrect values are only used to update the boundary of the grid but do not influence the divisional surface.

All in all, the call to the \texttt{unidirHierarchize}-routine calculates the hierarchical surpluses of divisional surface \( \mathcal{H}(G(x_0, k, \ell)) \) correctly while, in general, it does not for the boundary. If the grid is at the highest level \( k = \ell \), however, the updates of all grid points, including those at the boundary of the poles, can be performed correctly. Hence, the call \texttt{unidirHierarchize}(G(0, k, \ell)) also hierarchizes the boundary correctly. For \( k < \ell \) we copy the original values of the boundary back at Step 10. Hence the invariant holds.

Finally, after the divisional surface \( \mathcal{H}(G(x_0, k, \ell)) \) has been hierarchized by Step 8 these values are not altered at any other point of Algorithm 6.1. Hence the whole grid is hierarchized correctly.

\[ \text{Theorem 6.4. Assuming a tall cache of size } M \in \omega \left( B^d \right) \text{ and, in particular, } B \leq \frac{1}{2^d} \cdot \sqrt[4]{\frac{M}{6}} \text{ as well as } M = o \left( |C_\ell| \right) = o \left( \left( 2^\ell + 1 \right)^d \right) \text{ and that } C_\ell \text{ is stored in row-major order, Algorithm 6.1 causes at most } \]

\[ |C_\ell| \cdot \left( \frac{1}{B} + o \left( \frac{1}{B^{\frac{d}{M}}} \right) \right) \quad (6.1) \]

\text{cache misses in the cache-oblivious model.}

The cache-oblivious model assumes the best possible cache replacement strategy is used. Instead of the best possible cache replacement
strategy the following strategy, which can only deteriorate the complexity of the algorithm, is analyzed in the proof of Theorem 6.4. Initially load \( Z_{k_1} \) to internal memory for \( 1 \leq k_1 \leq \log_2 \left( \sqrt[3]{M/6} - 1 \right) \) and hold this set in internal memory during the execution of the whole algorithm. The analysis in Lemma 6.6 shows that these sets \( Z_{k_1} \) occupies at most half of the internal memory. Apply the LRU strategy to the other half of the internal memory.

We split the proof of the theorem into two lemmas depending on the size of the subgrid which is currently hierarchized. Lemma 6.5 provides a brute force analysis for the number of cache misses of Algorithm 4.2. This brute force analysis is used if the current subgrid is too large to fit into cache. Lemma 6.6 analyses the number of cache misses for small subgrids more carefully.

**Lemma 6.5.** For fixed \( x, k \) and \( \ell \), Algorithm 6.1 causes at most

\[
d \cdot 3 \cdot |☐(G(x, k, \ell))| + 4 |☐(G(x, k, \ell))| = O \left( \left( 2^{k_1} \right)^{d-1} \right).
\]

cache misses between and including Step 7 and Step 11. For this, \( M \geq 3B \) suffices.

**Proof.** As each \( Z_{k_1} \) is contiguous in memory, both copy operations together (Step 7 and Step 10) cause at most

\[
2 \cdot 2 \cdot |☐(G(x, k, \ell))| \leq 4 \cdot 2d \cdot \left( 2^{k_1} + 1 \right)^{d-1} = 8d \cdot \left( 2^{k_1} + 1 \right)^{d-1}
\]
cache misses. \( M \geq 2B \) suffices for copying.

It is left to analyze the costs of the \texttt{unidirHierarchize}-routine. This routine loops \( d \) times over \( ☐(G(x, k, \ell)) \). In each of these \( d \) passes, the routine updates each point of \( ☐(G(x, k, \ell)) \) exactly once. For each update, up to 3 different grid points need to be read and the updated value is stored at one of these positions. Hence, at most 3 cache misses can occur for the update of one grid point in each pass. In total, the number of cache misses is at most

\[
d \cdot 3 \cdot |☐(G(x, k, \ell))| \leq 3d \cdot 3d \cdot \left( 2^{k_1} + 1 \right)^{d-1} = 9d^2 \cdot \left( 2^{k_1} + 1 \right)^{d-1}.
\]

\( M \geq 3B \) was sufficient for that.

Adding both terms results in at most

\[
8d \cdot \left( 2^{k_1} + 1 \right)^{d-1} + 9d^2 \cdot \left( 2^{k_1} + 1 \right)^{d-1} \leq 17d^2 \cdot \left( 2^{k_1} + 1 \right)^{d-1} = O \left( \left( 2^{k_1} \right)^{d-1} \right)
\]
cache misses in total. For this analysis the data layout was irrelevant as we assumed a cache miss for each accessed item. \( \square \)
Lemma 6.6. Assume a tall cache of size \( M \in \omega (B^d) \) and, in particular, \( B \leq \frac{1}{2^d} \cdot \sqrt[6]{\frac{M}{6}} \). Furthermore, assume \( h_0 := \log_2 \left( \frac{\sqrt[6]{M/6} - 1}{6} \right) - 1 \), and that the set \( \bigcup_{k_1=1}^{h_0} Z_{k_1} \) is already in internal memory at the beginning of the algorithm (and is always kept there). In addition, assume that the LRU cache replacement strategy is applied to all blocks of internal memory that are not occupied by \( \bigcup_{k_1=1}^{h_0} Z_{k_1} \). Then, for fixed \( x, k \) and \( \ell \) with \( k_1 \leq h_0 \), the number of cache misses the call \( \text{hierarchizeRec}(G(x, k, \ell)) \) in Algorithm 6.1 (and all subsequent calls in the recursion) is less than

\[
\frac{(2^k - 1)^d}{B} + O \left( \frac{2^k}{d} \right) .
\]

Proof. The threshold \( h_0 \) was chosen such that \( G(x, k, \ell) \cup \bigcup_{k_1=1}^{h_0} H_{k_1} \) fits into cache. Assuming a row-major layout, the number of cache lines occupied by \( G(x, k, \ell) \) is less or equal to

\[
\left( \left\lfloor \frac{2^{k_1} + 1}{B} \right\rfloor + 1 \right) \cdot \left( \frac{2^{k_1} + 1}{B} \right)^{d-1} \leq \frac{\left( \frac{2^{k_1} + 1}{B} \right)^d}{B} + 2 \cdot \left( \frac{2^{k_1} + 1}{B} \right)^{d-1} \leq
\]

\[
\frac{\left( \frac{\sqrt[6]{M/6} + 1}{B} \right)^d + 2 \cdot \left( \frac{\sqrt[6]{M/6} + 1}{B} \right)^{d-1}}{B} = \frac{M}{6 \cdot B} + 2 \cdot \left( \frac{\sqrt[6]{M/6}}{B} \right)^{d-1} \leq \left( B \leq \frac{1}{2^d} \cdot \sqrt[6]{\frac{M}{6}} \right) \frac{M}{6 \cdot B} + \frac{M}{6 \cdot d \cdot B} \leq \frac{1}{3} \cdot \frac{M}{B} .
\]

As each \( Z_{k_1} \) is contiguous in memory, the number of cache lines occupied by \( \bigcup_{k_1=1}^{h_0} H_{k_1} \) is at most

\[
\sum_{k_1=1}^{h_0} |H_{k_1}| = \sum_{k_1=1}^{h_0} \left\lfloor \frac{2d \cdot (2^k + 1)^{d-1}}{B} \right\rfloor \leq
\]

\[
\leq h_0 + \frac{2d}{B} \cdot \sum_{k_1=1}^{h_0} \left( 2 \cdot 2^k \right)^{d-1} \leq
\]

\[
\leq h_0 + \frac{2d \cdot 2^{d-1}}{B} \cdot \frac{(2^{d-1})^{h_0+1} - 2^{d-1}}{2^{d-1} - 1} \leq \leq \left( B \leq \frac{1}{2^d} \cdot \sqrt[6]{\frac{M}{6}} \right) \frac{\sqrt[6]{M/6} + 2d}{2d \cdot B} \cdot \frac{2^{d-1}}{B} \cdot \left( \frac{\sqrt[6]{M/6}}{B} \right)^{d-1} \leq
\]

\[
\leq \frac{\sqrt[6]{M/6}}{B} \left( \frac{\sqrt[6]{M/6} + 2d}{2d \cdot B} \cdot \left( \frac{\sqrt[6]{M/6}}{B} \right)^{d-1} \right) = \frac{M}{12d \cdot B} + \frac{M}{3 \cdot B^2} \leq \frac{1}{2} \cdot \frac{M}{B} .
\]
As all cache lines of \( G(x, k, \ell) \cup \bigcup_{k_1=1}^{h_0} H_{k_1} \) fit into cache, \( G(x, k, \ell) \) can be hierarchized by loading it once, performing the updates and copy operations, and storing the final results. Recall that \( G(x, k, \ell) \) is always kept in cache while a LRU cache replacement strategy is employed for the part of the cache that stores \( G(x, k, \ell) \). As no other cache lines are touched during the call \( \text{hierarchizeRec}(G(x, k, \ell)) \), only cache misses to load \( G(x, k, \ell) \) occur. Assuming a row-major order, loading \( G(x, k, \ell) \) causes at most
\[
\left( \left\lceil \frac{2k_1 + 1}{B} \right\rceil + 1 \right) \cdot (2k_1 + 1)^{d-1} = \left( \frac{2k_1 - 1}{B} \right)^d + \mathcal{O} \left( 2k_1 \right)^{d-1}
\]
cache misses.

Proof of Theorem 6.4. The complexity of divide and conquer algorithms assumes the best cache replacement strategy is used. Thus, using any particular cache replacement strategy in the analysis yields an upper bound in the cache-oblivious sense. We analyze a cache replacement strategy that is limited as follows: Assume the algorithm always holds the \( H_{k_1} \) for \( 1 \leq k_1 \leq h_0 := \log_2 \left( \sqrt[2]{M/6} - 1 \right) + 1 \) in internal memory. Besides that, the LRU cache replacement strategy is used. To load \( \bigcup_{k_1=1}^{h_0} H_{k_1} \) initially, the algorithm causes at most (compare to Equation 6.2)
\[
\sum_{k_1=1}^{h_0} |H_{k_1}| = \mathcal{O} \left( M^{\frac{1}{d}} \right) + \mathcal{O} \left( \frac{M^{\frac{d-1}{d}}}{B} \right) = \mathcal{O} \left( M^{\frac{d-1}{d}} \right)
\]
cache misses.

We analyze the algorithm in two steps, \( k_1 = h_0 \) and \( k_1 > h_0 \). For \( k_1 = h_0 \), we can apply Lemma 6.6. As the interior of the subgrid \( G(x, k, \ell) \) contains \( (2k_1 - 1)^d \) vertices and the different subgrids are disjoint, the number of subgrids of level \( k \) is bounded above by
\[
\frac{|C_\ell|}{(2k_1 - 1)^d}.
\]
Therefore and by Lemma 6.6, level \( k_1 = h_0 \) of the recursion and everything below it cause at most
\[
\frac{|C_\ell|}{(2k_1 - 1)^d} \cdot \left( \left( \frac{2k_1 - 1}{B} \right)^d + \mathcal{O} \left( 2k_1 \right)^{d-1} \right) = |C_\ell| \cdot \left( \frac{1}{B} + \mathcal{O} \left( \frac{1}{\sqrt{M}} \right) \right)
\]
cache misses assuming \( C_\ell \) is stored in row-major order.
It is left to handle the levels of the recursion when \( h_0 < k_1 \leq \ell_1 \). Again, the number of subgrids of level \( k \) is at most \( |C_k| / \left(2^{k_1} - 1\right)^d \). Hence, by Lemma 6.5, the number of cache misses to hierarchize the divisional surface of all subgrids of level \( k \) is at most \( |C_k| \cdot \Theta \left(\frac{1}{2^{k_1}}\right) \). To get the number of cache misses for all levels above the threshold, just sum,

\[
\sum_{k_1 = h_0}^\ell |C_k| \cdot \Theta \left(\frac{1}{2^{k_1}}\right) = |C_k| \cdot \Theta \left(\frac{1}{2^{h_0}}\right) = |C_k| \cdot \Theta \left(\frac{1}{\sqrt[\ell]{M}}\right).
\]

(6.5)

Adding Equation 6.3, Equation 6.4 and Equation 6.5 yields

\[
\Theta \left(M \cdot \frac{d^2}{\sqrt[\ell]{M}}\right) + |C_k| \cdot \Theta \left(\frac{1}{B} + \Theta \left(\frac{1}{\sqrt[\ell]{M}}\right) + \Theta \left(\frac{1}{\sqrt[\ell]{M}}\right)\right) =
\]

(\(M = o(|C_k|)\))

\[
|C_k| \cdot \Theta \left(\frac{1}{B} + \Theta \left(\frac{1}{\sqrt[\ell]{M}}\right)\right).
\]

as upper bound for total number of cache misses of Algorithm 6.1.

The \( d \) factor hidden in the \( \Theta \)-notation is quadratic, i.e. the complexity of Algorithm 6.1 in the cache-oblivious model including the constant \( d \) is

\[
|C_k| \cdot \Theta \left(\frac{1}{B} + \Theta \left(\frac{d^2}{\sqrt[\ell]{M}}\right)\right).
\]

This is implied by brute force analysis in Lemma 6.5 where the \( d^2 \) factor is absorbed by the \( \Theta \)-notation. In Lemma 6.6, the absorbed factor depends linearly on \( d \) and comes from

\[
\left(2^{k_1} + 1\right)^{d-1} = \left(2^{k_1} - 1\right)^{d-1} + \Theta \left(d \cdot \left(2^{k_1}\right)^{d-1}\right).
\]

Parallelizing Algorithm 6.1 is straightforward as all subgrids of the same level of the recursion can be handled in parallel. Hence, to work with \( P = (2^d)^p \) processors call hierarchizeRec(\( \cdot \)) with the input values for each of the \( P \) subgrids of level \( k_1 = \ell_1 - p \). Then finish the coarsest \( p \) levels in serial or with less parallelism. We have seen in the proof of Theorem 6.4 that the coarse levels only contribute to the lower order term.

For \( B > 1 \) and a row or column layout, there were always dimensions in which blocks that store the vertices of the divisional surface and boundary also contain vertices of the interior of the current subgrid and outside of the current subgrid. For \( B = 1 \), it is possible to exactly load the divisional surface and boundary to internal memory without polluting the internal memory with other vertices. This enables us to increase
the threshold \( h_0 \) to \( h_0 := \log_2 \left( \mathcal{O} \left( d^{-1} \sqrt{M} \right) \right) \) when changing the \( Z_{k_1} \) to store \( \bigcup_{k_1=1}^{h_0} \mathcal{H}(G(x, k, \ell)) \). The latter only increases the amount of additional out-of-place memory by a factor of \( 3/2 \). In the analysis of Lemma 6.6 it is then sufficient if two copies of \( \bigcup_{k_1=1}^{h_0} \mathcal{H}(G(x, k, \ell)) \) can be kept in memory. One copy stores the original values as the \( Z_{k_1} \) do, and the the other copy is used to update the grid points. Performing the rest of the analysis as before yields \( |C_\ell| \cdot \left( 1 + \mathcal{O} \left( \frac{1}{d-1} \sqrt{M} \right) \right) \) for the complexity of Algorithm 6.1 and hence matching upper and lower bounds.

### 6.3 Lower Bound

#### 6.3.1 Overview of the Lower Bound

Let us now address the lower bound. While the divide and conquer algorithm is analyzed in the cache-oblivious model, the lower bound is proven in the cache-aware external memory model as this only strengthens the lower bound. Recall that the trivial scanning lower bound of \( |C_\ell|/B \) already matches the constant of the leading term of the cache misses of the divide and conquer algorithm. Hence, all subsequent analysis proves a non-trivial lower order term. The lower bound requires the assumption \( |C_\ell| \in \omega(M^{d+1}) \) guaranteeing that the input is significantly larger than the cache but does not need the tall cache assumption.

The proof of the lower bound builds upon the assumption that the algorithm can only compute linear combinations and not arbitrary algebraic or transcendental functions of the input. This assumption is natural as hierarchization is indeed a linear operation and can be formulated as multiplying the vector containing the input values of the grid points with a sparse matrix. The lower bound for hierarchization uses a similar argument as the lower bound for stencil computations (see Section 3.5) and builds upon the technique developed by Hong and Kung [HK81]: an arbitrary algorithm is split into rounds and then a kind of isoperimetric result (or dimensionality or rank argument) is applied. For the stencil computations, the algorithm was split into rounds by the number of I/Os it performs and then the progress per round was upper bounded with an isoperimetric argument. For hierarchization, the algorithm is split into rounds by the progress it makes, i.e. the number of correctly hierarchized grid points. Then, the isoperimetric result gives a lower bound for the number of data items the algorithm has to access to hierarchize these grid points. As we derive the order of the lower order term but not its constant, as in the case for stencils, a rough isoperimetric estimate suffices.

The isoperimetric result builds upon the assumption that the algorithm is linear. If a linear algorithm maps onto (i.e., surjectively) the set of output vertices of a certain round, then the set of vertices that is
accessed during that round, including input vertices and intermediate results, has to be at least as large as the output set.

We first prove the lower bound for $B = 1$ and then generalize it to arbitrary $B$ by the observation that one I/O can transfer at most $B$ vertices at once. As a result, the lower bound applies to any grid layout and is not limited to row- or column-major order.

### 6.3.2 Derivation of the Lower Bound

As a lower bound of $|C_\ell|/B$ is given trivially by the need to scan the input, all subsequent analysis derives the leading lower order term, i.e. the leading term of the non-compulsory I/Os, in Theorem 6.7. To prove the non-trivial lower bound for the term of the non-compulsory I/Os, we need to assume that the grid is significantly larger than the internal memory. To be precise, we need the technical assumption

$$|C_\ell| \in \omega \left( M \frac{d+1}{d-1} \right).$$

This assumption is slightly stronger than the usual tall cache assumption $M \in o(|C_\ell|)$.

**Theorem 6.7.** Assuming that the grid is significantly larger than the cache, i.e. $|C_\ell| = \omega(M \frac{d+1}{d-1})$, any algorithm $A$ that computes the hierarchical surpluses of the component grid $C_\ell$ incurs at least the following number of cache misses:

$$|C_\ell| \cdot \left( \frac{1}{B} + \Omega \left( \frac{1}{B \frac{1}{\sqrt{M}}} \right) \right).$$

In the proof of the lower bound we are satisfied if the algorithm hierarchizes all interior grid points of $C_\ell$ correctly. The boundary vertices do not need to be hierarchized necessarily. We first provide the lower bound for $B = 1$ and then generalize it to arbitrary $B$.

Furthermore, during the derivation of the lower bound we only count cache misses for reading data. This assumption is justified as it can only reduce the number of cache misses and hence weaken the lower bound. Fetching a cache line to store data in it is assumed to be free of charge. Hence, we can assume that the algorithm analyzed by the lower bound works out-of-place with two copies of the grid, one which initially stores the input values and one to store the output. For the initial analysis we assume that loading an input vertex causes 1 cache miss. Loading intermediate results, called variables, incurs 2 cache misses. This assumption is going to be dropped at the end of the proof of the lower bound.

It also needs to be assumed that the control flow of an algorithm only depends on the size and structure of the input grid, i.e., the parameters $d$ and $\ell$, but not on the actual numerical values that are stored at the grid points. For the proof of the lower bound, the control flow of an algorithm has to be independent of the numerical input values, i.e., the
values stored at the grid points. One way to enforce this is to disallow comparisons of the numerical input values. This assumption does not exclude any correct algorithm as argued in [BBF+10]. The assumption is needed as we are going to alter the numerical input values while we assume that the splitting of an algorithm into its rounds remains identical.

Assume $A$ is an arbitrary algorithm hierarchizing all interior grid points of $C_\ell$. Divide $A$ into rounds by the number of hierarchized grid points it writes back to memory. The rounds are chosen such that $A$ stores $|R| = 3M$ output points in each round, except possibly the last. As we assume that the algorithm can compute arbitrary linear combinations but no algebraic or transcendental functions of the input, we can regard $A$ as a linear map from the input to the output grid, $A : C_{\text{in}}^\ell \rightarrow C_{\text{out}}^\ell$. We know that this map is onto as the inverse map dehierarchization, i.e. the transform from the hierarchical surpluses to the function values at the grid points, exists. As each round calculates the hierarchical surpluses of a subset of the grid, this yields that the input to each round has to be at least as large as the set that is output. As calculating the hierarchical surplus of a grid point does not only require the value of the grid point itself but also the values of its hierarchical predecessors, the input to a round has to be larger than the output. This enables us to prove the non-trivial lower order term.

As a first step, we want to know how many poles of a certain dimension are at least needed to cover the output vertices of one round. Therefore assume that $R$, the output vertices of one round, are embedded into the infinite grid $\mathbb{Z}^d$ by multiplying the coordinates of the grid points of $R$ by $2^\ell$.

**Lemma 6.8.** For $d \geq 2$ and any set $R \in \mathbb{Z}^d$, the sum over the number of grid points in the projections of $R$ is lower bounded by

$$\sum_{r=1}^{d} |\pi_r(R)| \geq d \cdot |R|^{(d-1)/d}.$$  

**Proof.** By induction on $d$. Base case $d = 2$: as $R \subseteq \pi_1(R) \times \pi_2(R)$, we have $|R| \leq |\pi_1(R)| \cdot |\pi_2(R)|$. We relax this to the real-valued optimization problem to minimize $a + b$ under the constraint $ab \geq |R|$ which has the well known solution $a = b = \sqrt{|R|}$. From this we conclude the statement of the lemma, $|\pi_1(R)| + |\pi_2(R)| \geq 2|R|^{1/2}$.

Induction $d - 1$ to $d$: we “slice” $R$ in direction $d$ by defining for $t \in \mathbb{Z}$ the $d - 1$ dimensional sets

$$R_t = \{x \in \mathbb{Z}^{d-1} \mid (x_1, \ldots, x_{d-1}, t) \in R\}.$$
We have \(|R| = \sum_{t \in \mathbb{Z}} |R_t|\). For \(r < d\) it holds that \(|\pi_r(R)| = \sum_t |\pi_r(R_t)|\) and \(|\pi_d(R)| \geq |\pi_d(R_t)| = |R_t|\) for all \(t\). By the inductive hypothesis, we have
\[
\sum_{r=1}^{d-1} |\pi_r(R_t)| \geq (d - 1) |R_t|^{(d-2)/(d-1)}
\]
We sum over \(t\) and relax from the integral to the real-valued optimization problem, where \(X\) stands for \(|\pi_d(R)|\) and \(\alpha_t\) denotes \(|\pi_r(R_t)|\):
\[
\text{minimize } X + (d - 1) \sum_t \alpha_t^{(d-2)/(d-1)}
\]
subject to \(\sum_t \alpha_t = |R|\) and \(\alpha_t \leq X\).

The function \(g(x) = x^{(d-2)/(d-1)}\) is a monotonically increasing concave function and hence it holds that \(g(x \cdot X) \geq x \cdot g(X)\). For the same reason, the objective function decreases if the “mass” of \(|R|\) is concentrated as much as possible. Hence, for fixed \(X\), the optimal value is given for
\[
\alpha_1 = \cdots = \alpha_{\lceil |R|/X \rceil} = X, \quad \alpha_0 = |R| - \lfloor |R|/X \rfloor \cdot X, \quad \text{all other } \alpha_t = 0.
\]
As
\[
g(|R| - \lfloor |R|/X \rfloor X) \geq (|R|/X - \lfloor |R|/X \rfloor)g(X),
\]
we can lower bound the objective function by
\[
X + (d - 1)(|R|/X)g(X) = X + (d - 1)|R|/d\sqrt[d-1]{X} =: f(X).
\]
Because the derivative \(\frac{d}{dX} f(X) = 1 - |R|/X^{d/(d-1)}\) is monotonically decreasing and 0 for \(X = |R|^{(d-1)/d}\), we can conclude that the minimal value is
\[
f(|R|^{(d-1)/d}) = |R|^{(d-1)/d} + (d - 1)|R|/|R|^{1/d} = d \cdot |R|^{(d-1)/d}.
\]

\[\square\]

**Corollary 6.9.** For any set \(R\) of the grid there is a dimension \(r\) such that at least \(|R|^{(d-1)/d}\) poles of dimension \(r\) contain points of \(R\).

**Proof.** By Lemma 6.8 there is at least one dimension \(r\) such that \(\pi_r(R) \geq |R|^{(d-1)/d}\) after \(R\) has been transferred to the infinite grid \(\mathbb{Z}^d\). This means that at least \(|R|^{(d-1)/d}\) poles of dimension \(r\) are needed to cover \(R\).

\[\square\]

For a grid point \(x \in G\) let \(x^{in} \in G^{in}\) and \(x^{out} \in C^{out}_t\) denote the grid points in the input respective output grid with the same coordinates as \(x\).
Similarly, for a set \( S \subset G \) let \( S^{in} \subset G^{in} \) and \( S^{out} \subset G^{out} \) denote the sets of the input respective output grid with the same grid points as \( S \).

Assuming a round of algorithm \( A \) is fixed, we divide all interior vertices of \( G \) that \( A \) accesses during this round into the following sets (given a fixed dimension \( r \)). Given these sets, we use Corollary 6.9 to prove further isoperimetric results.

- **Input vertices and variables in cache in that round:**
  - \( \text{I}_{\text{dir}}. \) - direct input: \( x_{\ell,i}^{in} \) for which \( x_{\ell,i}^{out} \) is output in that round.
  - \( \text{I}_{\text{ind}}. \) - indirect input: \( x_{\ell,i}^{in} \) for which \( x_{\ell,i}^{out} \) is not output in that round.
  - \( \text{V} \) - variables: any intermediate results.

- **Output vertices (hierarchical surpluses written to external memory):**
  - \( \text{O}_{\text{dir}}. \) - direct output: \( x_{\ell,i}^{out} \) for which \( x_{\ell,i}^{in} \) is in cache in that round.
  - \( \text{O}_{\text{ind}}. \) - indirect output: \( x_{\ell,i}^{out} \) for which \( x_{\ell,i}^{in} \) is not in cache in that round.

- **\( P_{\text{all}} \) - a representative for each pole in direction \( r \) of output vertices:**
  \( P_{\text{all}} = \pi_{\text{P}} \cap \text{O}_{\text{dir}} \cup \text{O}_{\text{ind}}. \)
  - \( \text{P}_{\text{dir}}. \) - direct pole: all output vertices of the pole are direct outputs.
    \( \text{P}_{\text{dir}} = \left\{ z \in P_{\text{all}} : \frac{1}{|x|} x_{\ell,i}^{out} \in \text{O}_{\text{ind}} \text{ such that } \pi_{\text{P}} \left( x_{\ell,i}^{out} \right) = z \right\}. \)
  - \( \text{P}_{\text{ind}}. \) - indirect poles: at least one output vertex of the pole is an indirect output.
    \( \text{P}_{\text{ind}} = \left\{ z \in P_{\text{all}} : \exists x_{\ell,i}^{out} \in \text{O}_{\text{ind}} \text{ such that } \pi_{\text{P}} \left( x_{\ell,i}^{out} \right) = z \right\}. \)

To prove isoperimetric inequalities for these sets, we show that the hierarchization algorithm is surjective for certain combinations of the above sets. To do so, we define a dependency graph and give topological sortings of that dependency graph that enable us to set the input values such that we achieve the desired outputs. Therefore, recall the definitions from Section 4.9. In particular, recall that the hierarchical surplus of \( x_{\ell,i} \) is determined by the initial value of the grid points \( x_{\ell,(i+a)} \) for \( a \in \{-1,0,1\}^d \) as given in Equation 4.6. We use that to build a (acyclic) dependency graph \( H = (V,E) \) for the hierarchization algorithm. The vertices of the graph are the vertices of the component grid, \( V = C_\ell. \)

There is a directed edge \( \left( x_{\ell,i}, x_{k,j} \right) \in E \) if \( x_{\ell,i}^{in} \) is part of the sum that deter-
mines the value of $x_{k,j}^{\text{out}}$. We do, however, explicitly exclude the self edge $(x_{k,j}, x_{k,j})$ and hence the edge set $E$ is limited to $E \subseteq ((H \setminus \{y\}) \times H)$.

$$E = \left\{ (x_{\ell,i}, x_{k,j}) : \exists a \in \{-1,0,1\}^d, a \neq 0 : (\ell,i) = (k,j + a) \text{ in simple form} \right\}. \quad (6.8)$$

If $a \neq 0$ in Equation 4.6, then $x_{\ell, (i+a)}$ is a (possibly transitive, i.e. predecessor of a predecessor) hierarchical predecessor of $x_{\ell,i}$. Hence, as detailed in Section 4.9, there exists $(k,j)$ such that $x_{\ell,(i+a)} = x_{k,j}$ and $k \leq \ell$, $k_r < l_r$ for some dimension $r$. Therefore ordering the vertices $x_{k,j} \in C_\ell = V$ by the sum of their level-vector $\|k\|_1$ gives a topological sorting of $H$.

**Lemma 6.10.** For any round and any dimension $r$: $|V| \geq |P_r^{\text{ind}}|$.

**Proof.** For each of the indirect poles of the round, choose any of the indirect outputs $x_{\ell,i}^{\text{out}}$ and denote this set $S^{\text{out}}$. Assume that all input except $S^{\text{in}}$ is set to 0. We want to show that the hierarchization algorithm $\mathcal{A}: \mathbb{R}^{|S^{\text{in}}|} \rightarrow \mathbb{R}^{|S^{\text{out}}|}$ is onto (surjective).

To achieve a certain output $S^{\text{out}}$, set the input values $S^{\text{in}}$ in the topological sorting of the hierarchization dependency graph given by the level-sum of the input vertices. The topological sorting guarantees that a value which is set influences only output vertices whose corresponding input vertex is set later. The self edge corresponding to $a = 0$ in Equation 4.6 excluded explicitly in Equation 6.8 guarantees that the value of a input vertex influences its output value. The exact value which is given to the input vertex depends on the values set previously. In summary, choosing the values of $S^{\text{in}}$ in the topological order given by the level-sum of the vertices enables us to construct a set of input values that maps to the desired output configuration.

As the algorithm calculating the hierarchical surpluses has to work for any input values, it has to work in particular for those specified above. The vertices that are in memory in the current round are $I_{\text{dir}}, I_{\text{ind.}}$ and $V$. As we have only allowed nonzero values for input vertices that correspond to indirect outputs, all direct inputs $I_{\text{dir.}}$ and all indirect inputs $I_{\text{ind.}}$ have been set to constant value 0. Hence, $\mathbb{R}^{|I_{\text{dir.}}|+|I_{\text{ind.}}|+|V|} \simeq \mathbb{R}^{|V|}$. As the mapping $\mathcal{A}: \mathbb{R}^{|S^{\text{in}}|} \rightarrow \mathbb{R}^{|S^{\text{out}}|}$ is onto, so has to be the mapping $\mathcal{A}: \mathbb{R}^{|I_{\text{dir.}}|+|I_{\text{ind.}}|+|V|} \rightarrow \mathbb{R}^{|S^{\text{out}}|}$. As $\mathcal{A}$ is a linear map by assumption, $\dim(\mathbb{R}^{|V|}) \geq \dim(\mathbb{R}^{|S^{\text{out}}|})$ and hence $|V| \geq |S^{\text{out}}| = |P_r^{\text{ind.}}|$. \hfill $\square$

**Lemma 6.11.** For any round and any dimension $r$: $|I_{\text{ind.}}| + |V| \geq |O_r^{\text{ind.}}| + |P_r^{\text{dir.}}|$.

**Proof.** Similar to the proof of Lemma 6.10. We want to show that the mapping $\mathcal{A}: \mathbb{R}^{|K^{\text{in}}|} \rightarrow \mathbb{R}^{|S^{\text{out}}|}$ is onto. The set $S^{\text{out}}$ upon which we want to
map onto is: choose all of the indirect outputs as well as one of the output points of smallest level in dimension $r$ for each of the direct poles in direction $r$. In particular, $|S^{\text{out}}| = |O_{\text{ind.}}| + |P_{\text{dir.}}|$. We set all input to 0 except for $K^{\text{in}}$: input vertices $x_{\ell,i}^{\text{in}}$ corresponding to indirect outputs $x_{\ell,i}^{\text{out}}$ (need to be covered by variables) and one of the hierarchical predecessors of the chosen vertex of each direct pole (going to be covered by indirect input). Note that such a hierarchical predecessor always exists as we have limited the set of output vertices to be in the interior of the grid $C_\ell$.

A topological sorting of the hierarchization dependency graph $H$ is given by the following two step procedure: first (subordinate), sort all poles of $H$ in direction $r$ by increasing level in dimension $r$ and second (dominant) sort all poles by the sum of their level vector disregarding dimension $r$. To see that this gives a topological sorting, consider an edge $(x_{\ell,i}, y_{k,j}) \in E$. As we excluded the self edge in $E$, it follows from Equation 6.8 that $\ell \leq k$ and $|\ell|_1 < |k|_1$. Hence there has to be a dimension $s$ for which $l_s < k_s$. If there is an $s \neq r$ for which $l_s < k_s$ holds, then the dominant sorting step places $x_{k,i}$ before $y_{k,j}$ and hence this edge is in order with the topological sorting. If there exists no such $s \neq r$, then $l_s = k_s \forall s \neq r$ and $l_r < k_r$. As $a_s \in \{-1, 1\}$ would imply $l_s < k_s$, it also follows from Equation 6.8 that $a_s = 0 \forall s \neq r$. Hence, $x_{k,i}$ and $y_{k,j}$ are part of the same pole in direction $r$ and are hence sorted by the subordinate sorting step which places $x_{k,i}$ before $y_{k,j}$. Therefore, the order is topological.

The topological sorting guarantees that a value which is set influences only output vertices whose corresponding input vertex is set later. The self edge corresponding to $a = 0$ in Equation 4.6 and excluded explicitly in Equation 6.8 guarantees that the value of a input vertex influences its output value. This enables setting the values of all indirect outputs. The output value of the vertex of smallest level for a direct pole, say $y_{k,j}$, can be manipulated by its hierarchical predecessor which is in $K^{\text{in}}$, say $x_{\ell,i}$. But we have to make sure that no vertex that is in between $x_{\ell,i}$ and $y_{k,j}$ in the topological order also influences the value of $y_{k,j}$. The only vertices between $x_{\ell,i}$ and $y_{k,j}$ in the given two step topological sorting are in the same pole as $x_{\ell,i}$ and $y_{k,j}$. Of the same pole, however, there is no nonzero input except $x_{\ell,i}$ and hence the value of $y_{k,j}^{\text{out}}$ cannot be altered any more after we have determined it by choosing $x_{\ell,i}^{\text{in}}$. Hence, we have shown that the mapping $A : \mathbb{R} |K^{\text{in}}| \rightarrow \mathbb{R} |S^{\text{out}}|$ is onto.

As the algorithm calculating the hierarchical surpluses has to work for any input values, it has to work in particular for those specified above. The vertices that are in memory in the current round are $I_{\text{dir.}}, I_{\text{ind.}}$ and $V$. All vertices of $I_{\text{dir.}}$ have been set to the constant value 0. Hence, we have $\mathbb{R} |I_{\text{dir.}}| + |I_{\text{ind.}}| + |V| = 0$. As the mapping $A : \mathbb{R} |S^{\text{in}}| \rightarrow \mathbb{R} |S^{\text{out}}|$
is onto, so has to be the mapping \( A : \mathbb{R}^{|I_{\text{dir}|} + |I_{\text{ind}|} + |V|} \rightarrow \mathbb{R}^{|S^\text{out}|} \). As \( A \) is linear map, this yields \( |I_{\text{ind}|} + |V| \geq |S^\text{out}| = |O_{\text{ind}|} + |P_{\text{dir}|} \).}

\[ \tag{6.9} \]

Lemma 6.12. For any round and any dimension \( r \):
\[
\big| \text{cache misses} \big| \geq |O_{\text{all}|} + |P_{\text{all}|}^r| - 2M .
\]

Proof. First assume the internal memory would be empty at the beginning of the round. Then, all input to the round, \( I_{\text{dir}} \), \( I_{\text{ind}} \), and \( V \), would need to be read at least once during the round. Recall that we currently assume that fetching a variable from external memory causes 2 cache misses. This assumption is going to be dropped at the end of the proof of Theorem 6.7. Using Lemma 6.10 and Lemma 6.11 the number of cache misses, counting cache misses for variables twice, would be at least \( |I_{\text{dir}|} + |I_{\text{ind}|} + 2 \cdot |V| \geq |O_{\text{dir}|} + (|O_{\text{ind}|} + |P_{\text{dir}|}) + |P_{\text{ind}|} = |O_{\text{all}|} + |P_{\text{all}|} \) per round. The internal memory can hold at most \( M \) entries at the beginning of a round. In the worst case, all entries would be variables that would otherwise cost 2 cache misses each. Hence, the number of cache misses that a round causes is at least: \( |O_{\text{all}|} + |P_{\text{all}|} - 2M \). \\

Proof of Theorem 6.7. First assume \( B = 1 \). We choose the size of a round, i.e. the number of output vertices of a round, to be \( |O_{\text{all}|} = (2d \cdot M)^{d/(d-1)} \) for all except the last round which may have less outputs. By Corollary 6.9, for each round there exists a dimension \( r \) such that the output set consists of at least \( |P_{\text{all}|} \geq |O_{\text{all}|}^{(d-1)/d} = 2d \cdot M \) poles in dimension \( r \). By Lemma 6.12, the number of cache misses per round (counting misses for variables twice) is hence at least \( |O_{\text{all}|} + |P_{\text{all}|} - 2M = (2d \cdot M)^{d/(d-1)} + 2(d - 1) \cdot M \). Hence, the number of cache misses (counting misses for variables twice) that any algorithm has to cause is at least (denote the interior vertices of \( C_\ell \) by \( \text{int}(C_\ell) \))
\[
\left\lfloor \frac{\left| \text{int}(C_\ell) \right|}{O_{\text{all}}} \right\rfloor \cdot (\lfloor |O_{\text{all}|} + |P_{\text{all}|} \rfloor - 2M) \geq
\]
\[
|C_\ell| \cdot \frac{\left| \text{int}(C_\ell) \right|}{|C_\ell|} \cdot (\lfloor |O_{\text{all}|} + |P_{\text{all}|} \rfloor - 2M) - (|O_{\text{all}|} + |P_{\text{all}|} \rfloor - 2M) =
\]
\[
|C_\ell| \cdot \left( \frac{2^{l_1} - 1}{2^{l_1} + 1} \right)^d \cdot \frac{(2d \cdot M)^{d/(d-1)} + 2(d - 1) \cdot M}{(2d \cdot M)^{d/(d-1)}} -
\]
\[
- \Theta \left( (d \cdot M)^{d/(d-1)} \right) =
\]
\[
|C_\ell| \cdot \left( 1 - \Theta \left( \frac{1}{2^{l_1}} \right) \right) \cdot \left( 1 + \frac{2(d - 1)}{2d \cdot \sqrt{2d} \cdot \sqrt{d - 1} \sqrt{M}} \right) -
\]
\[
- \Theta \left( \frac{M \cdot d + 1}{M \cdot d - 1} \right) =
\]
\[
\begin{aligned}
&\frac{(6.6)}{64} |C_\ell| \cdot \left(1 - \Theta \left(\frac{1}{\sqrt[\ell]{|C_\ell|}}\right)\right) \cdot \left(1 + \Theta \left(\frac{1}{\sqrt{d-1}M}\right)\right) - o\left(\frac{|C_\ell|}{d-1\sqrt{M}}\right) = \\
&\frac{(6.6)}{64} |C_\ell| \cdot \left(1 - o \left(\frac{1}{M^{d+1} \sqrt{d}}\right)\right) \cdot \left(1 + \Theta \left(\frac{1}{\sqrt{d-1}M}\right)\right) = \\
&= |C_\ell| \cdot \left(1 + \Theta \left(\frac{1}{d-1\sqrt{M}}\right)\right).
\end{aligned}
\]

As we still assume \(B = 1\), \(|C_\ell|\) cache misses have to happen for reading the input. Hence, all cache misses caused by the variables are accounted for in the term \(\Theta(1/\sqrt{d-1}M)\). Dropping the assumption that loading variables incurs two cache misses instead of one, can at most halve this term but does not change its order. The result for general \(B \geq 1\) follows from the fact that a cache miss loads at most \(B\) vertices.

The lower order term reads \(\Theta \left(\frac{1}{\sqrt{d}} \cdot \frac{1}{\sqrt{d-1}M}\right)\) when the constant \(d\) is stated explicitly. This can be derived from Equation 6.9.

6.4 Discussion

The divide and conquer algorithm presented in this chapter provides a new algorithmic approach to sparse grid problems. It avoids the \(d\) global phases of the unidirectional principle and, by doing so, significantly reduces the number of cache misses in the cache-oblivious model given a tall cache of size \(M \in \omega(B^d)\). This algorithmic idea can also be beneficial for various other scenarios and this section discusses three such generalizations of the algorithmic approach. If the cache is smaller, i.e., of size \(M = \omega(B^r)\) for some \(r \in \{1, \ldots, d\}\), then the algorithm can be adapted to work on \(r\)-dimensional subproblems and merge \(r\) instead of \(d\) phases of the unidirectional principle. Also, the divide and conquer approach is neither limited to hierarchization, nor to component grids, let alone isotropic component grids. Furthermore, a cache-aware version of the presented divide and conquer algorithm is an alternative to the discussed cache-oblivious version.

First, let us address a cache-aware divide and conquer approach. The cache-aware approach can work in two stages and hence avoid recursion. The first stage would hierarchize a subgrid with the unidirectional algorithm when it essentially fits into cache, instead of recursing all the way to the base case of \(\ell = 1\). All grid points not hierarchized in this first stage, would then be hierarchized by a single call to the unidirectional hierarchization algorithm for sets, Algorithm 4.2. The first, fine-grain stage is similar to Lemma 6.6 and would work on subgrids that completely fit into cache. Hence, the interior of these subgrids can be hierarchized efficiently using the unidirectional algorithm. After finishing a subgrid, the algorithm would reestablish the original values for grid points at the boundary. The refinement level of the subgrids can be
chosen (slightly) larger than the threshold $h_0$ in Lemma 6.6, as only the subgrid itself and an additional copy of its boundary would need to fit into memory. After that fine grain level is finished, all unhierarchized grid points would be hierarchized by one application of the unidirectional hierarchization algorithm for sets, Algorithm 4.2. This combines multiple levels of the recursion of the divide and conquer algorithm in one go and further decreases the number of cache misses. Of course, the coarse grain stage can be further refined. For two levels of the memory hierarchy, this cache-aware approach would have a lower complexity than the divide and conquer algorithm and can, furthermore, work completely with a LRU replacement strategy. As a drawback, the algorithm would need to have the cache size as a parameter. The tall cache assumption $M \in \omega (B^d)$ is also essential for a cache-aware approach, as $B^d$ is the size of the smallest subgrid that can be chosen, assuming a standard column-major layout.

Second, the divide and conquer approach can also be easily generalized to other kinds of sparse grids, including anisotropic component grids as well as regular or adaptive sparse grids. For regular and adaptive sparse grids, the subgrids have fewer interior vertices per boundary vertex, which degrades the performance of the algorithm. For regular sparse grids, the sum in Equation 6.5 does not decay exponentially and hence influences the leading term. The general algorithmic approach of divide and conquer can, however, be applied to any kind of sparse grid and sparse grid algorithm. For a divide and conquer approach to be efficient, it is essential that the (orthant) subgrids are stored in contiguous memory, at least with respect to one dimension as it is the case in a row- or column-major layout. rSG [Jac14] provides such a data layout for regular sparse grids as it stores the 1-dimensional poles one after another in contiguous memory. Furthermore, the idea of avoiding the unidirectional principle globally but applying it locally is neither limited to the piecewise-linear basis nor to hierarchization.

Third, the strong tall cache assumption of $M = \omega (B^d)$ is necessary as the presented algorithm merges all $d$ phases of the unidirectional principle to a global one. To merge all $d$ phases, the algorithms works on $d$-dimensional subproblems whose size is at least $B^d$ if the subproblems contain at least $B$ items per dimension. If the cache is smaller, i.e., of size $M = \omega (B^r)$ for some $r \in \{1, \ldots, d\}$, then the algorithm can work on $r$-dimensional subproblems and merge $r$ instead of $d$ phases of the unidirectional principle to a single one. This enables us to work with a smaller cache and decrease the number of global sweeps over the data to $\lceil d/r \rceil$ compared to the $d$ sweeps of the unidirectional algorithm.
Hierarchization describes the base change from the full grid basis to the hierarchical basis. The hierarchization algorithm is one of the crucial algorithms for sparse grids as the hierarchical basis is one of the key components that enables sparse grids to reduce the curse of dimensionality. Furthermore, hierarchization is a prototypical algorithm for sparse grids, i.e., the techniques applied to decrease the cache misses of the hierarchization algorithm are also likely to be beneficial for other sparse grid algorithms. Up to now, all implementations of the hierarchization algorithm, and most sparse grid algorithms in general, have taken advantage of the unidirectional principle which performs $d$ global passes over the data. As a consequence, any algorithm implementing the unidirectional principle is inherently cache inefficient, as the whole grid is loaded essentially $d$ times from main memory to cache as soon as the grid is significantly larger than the cache. In addition, Chapter 5 has discussed an implementation of the unidirectional hierarchization algorithm that runs within a factor of 1.5 of this unidirectional lower bound. Hence, any further performance gains had to avoid the unidirectional principle and use novel algorithmic approaches.

This chapter has derived a divide and conquer approach that hierarchizes isotropic component grids $C_\ell$ with

$$|C_\ell| \cdot \left( \frac{1}{B} + \Theta \left( \frac{1}{\sqrt{dM}} \right) \right)$$

cache misses in the cache-oblivious model, given a tall cache of size $M \in \omega(B^d)$. With respect to the leading term, this is a factor $d$ less than the number of cache misses any unidirectional hierarchization algorithm has to cause. The number of cache misses was reduced by avoiding the unidirectional principle globally but applying it recursively to smaller subproblems.

In addition, the upper bound has been complemented with a lower bound of

$$|C_\ell| \cdot \left( 1 + \Theta \left( \frac{1}{d^{\frac{1}{2}} \sqrt{M}} \right) \right).$$

The lower bound is restricted to algorithms that can compute arbitrary linear combinations but no algebraic or transcendental functions of the input. This is a natural assumption, as hierarchization is indeed a linear operation. For $B = 1$, a small modification of the divide and conquer algorithm yields that the lower and upper bounds match.

Although this chapter has presented an algorithm whose complexity is optimal in the cache-oblivious model with respect to the leading term, the algorithm was not implemented. The algorithms involves many redundant computations and copy operations and uses the divide and
conquer approach all the way down to the base case of $\ell = 1$. As discussed in Section 6.4, a cache-aware version can have a simpler structure and, in addition, a lower complexity for two levels of the memory hierarchy. Hence, a cache-aware version of the presented algorithm offers an alternative for an implementation that is aimed to provide a proof of concept that the divide and conquer paradigm can be applied efficiently to sparse grid algorithms. In addition, we have also discussed in Section 6.4 that the algorithmic approach of merging several phases of the unidirectional principle to reduce the number of cache misses is also applicable if the cache does not satisfy the tall cache assumption $M = \omega \left( B^d \right)$.

Besides an implementation of the presented algorithm, future work can use the prototypical character of hierarchization and apply the divide and conquer paradigm to other sparse grid algorithms. One class of sparse grid algorithms which might benefit from a divide and conquer approach are so called UpDown schemes. UpDown schemes are used to implement more complicated algorithms, such as PDE solvers, directly in the hierarchical sparse grid basis.
COMMUNICATION SCHEMES

7.1 INTRODUCTION

Having memory efficient hierarchization algorithms at hand, this chapter addresses the communication step of the sparse grid combination technique. Consider solving a high dimensional PDE with the combination technique and assume for this chapter that the solution for each component grid has been computed on its own compute node of a HPC system. While the combination technique breaks the global communication requirements of conventional discretization approaches and lets us solve the original problem in parallel on the component grids, synchronization is still required between the time steps or at least every few time steps. In this synchronization step, each component grid is updated with the current value of the global function, the so called combination technique solution (see Equation 4.5). For that, the combination technique first has to assemble the combination technique solution from the component grid solutions (reduce). Then, the joint combination technique solution has to be distributed back to the component grids (broadcast). This reduce/broadcast step is the remaining communication bottleneck of the combination technique and requires a shrunken but global communication as illustrated in Figure 1.5.

Up to now, no special attention was paid to derive efficient algorithms for this remaining synchronization bottleneck. As discussed in Section 4.2, early communication schemes either use a farm of slaves to compute the component grid solutions and then assemble the sparse grid on a master or employ all-to-all communication. In summary, the communication step did not catch much attention as the applications were limited to 2 or 3 dimensions. For these low dimensional settings the number of component grids as well as their sizes stay reasonable.

For high dimensional settings, however, the number of component grids as well as their sizes increase significantly. In the setting of GENE discussed in Section 4.1, i.e. dimension $d = 5$ and sparse grid level $n = 10$, the combination technique would employ 1,876 component grids, each of a size of a couple of hundred kilobyte, assuming double precision. All component grids together would be about 1 gigabyte large and the sparse grid itself about 38 megabyte. The numbers grow rapidly as either the dimension or the refinement level of the discretization grow. For $d = 5$ and $n = 15$, there would be 9,626 component grids each of a size between 5 and 21 megabyte. In total, the component grids would take up about 100 gigabyte and the sparse grid itself about 3.3 gi-
gabyte. If \( d = 10 \) and \( n = 10 \), there would be 92,378 component grids each of a size between 44 and 154 megabyte. In total, the component grids would take up about 10 terabyte and the sparse grid itself about 134 gigabyte. Hence, if the combination technique is applied to high-dimensional settings, an efficient solution for the reduce/broadcast step is necessary.

This chapter derives two optimal communication strategies for the reduce/broadcast step, the remaining communication bottleneck of the combination technique. The first scheme, \textit{Sparse Grid Reduce}, is designed to minimize the number of communication rounds and does so by expanding each component grid locally to a sparse grid and performing one global \textit{AllReduce}-operation. Due to the single, global \textit{AllReduce}-operation, \textit{Sparse Grid Reduce} also sends few messages. As each component grid is expanded to the sparse grid before communicating it, the total as well as makespan communication volume of \textit{Sparse Grid Reduce} are, however, large. If the global \textit{AllReduce}-operation would assemble the sparse grid solution on a single node before broadcasting it, \textit{Sparse Grid Reduce} would be similar to the approach of computing the component grids on slave nodes and assembling the combination technique solution on the master [Gri92, GHR92, GHZ96]. Hence, \textit{Sparse Grid Reduce} can also serve as baseline. The second scheme, \textit{Subspace Reduce}, is optimal with respect to the total communication volume. \textit{Subspace Reduce} achieves minimal total volume by exploiting the hierarchical structure of sparse grids and splitting the communication step into substeps. Each substep performs an \textit{AllReduce}-operation for a hierarchical increment grid and exactly those nodes that contain the relevant increment grid as part of their component grid. As a consequence of splitting the problem into substeps, the number of rounds and messages sent by \textit{Subspace Reduce} increases.

After analyzing the schemes theoretically, their performance on HPC systems for 3 dimensions, the 5-dimensional setting of GENE, and an extended 10-dimensional setting is measured. Furthermore, this chapter presents a communication model which is well-suited to predict the cost of the communication step. Given performance characteristics of the employed HPC system, the model estimates lower and upper bounds for the runtime of the communication schemes. The model can also be applied to settings that are as yet out of scope due to the high computational demand and predict their communication feasibility for future HPC platforms.

In summary, the presented work is the first systematic study of the communication task of the sparse grid combination technique.

As the focus of this chapter is on the communication step of the combination technique, the model to predict the experiments disregards local computations. The communication model is described in detail in Section 7.3 and draws from all models for distributed computing presented
in Section 2.2. In particular, the additions performed by \textit{AllReduce} are not taken into account and we assume that the component grid solutions have already been multiplied with their respective coefficients from Equation 4.5.

The only communication operations carried out by the communication schemes are \textit{AllReduce}-operations and hence the optimality of the communication schemes depends on the optimality of the implementation of \textit{AllReduce}. For the analysis of the communication schemes it is assumed that \textit{AllReduce} has to work in two distinct phases, reduce and broadcast. For the upper bounds, i.e., the communication schemes, we assume further that each of these phases is implemented using binomial trees. While these assumptions facilitate the analysis of the communication schemes and the derivation of the lower bounds, they are not crucial for the optimality of the communication schemes. A communication scheme inherits optimality (assuming reduce and broadcast can be merged) with respect to a certain cost measure, if \textit{AllReduce} is implemented optimally (assuming reduce and broadcast can be merged) with respect to that cost measure and the communication scheme is optimal with respect to that cost measure for separate reduce and broadcast. Furthermore, the experiments are going to show that the model predicts the runtime of the algorithms well, even given the assumption that the reduce and broadcast phase are implemented separately by \textit{AllReduce}.

While \textit{Sparse Grid Reduce} can either work in the full grid basis or the hierarchical basis, \textit{Subspace Reduce} has to be applied to the hierarchical basis. As the solvers used on the component grids typically work in the full grid basis, this makes it necessary to hierarchize all component grids before the communication step and dehierarchize them thereafter. As hierarchization and dehierarchization can be performed locally for each component grid and as this work focuses on communication, these computations are neither taken into account in the model nor measured when the experiments are performed. In fact, while \textit{Subspace Reduce} has to hierarchize and dehierarchize the component grids, \textit{Sparse Grid Reduce}, and most likely any other communication scheme, also has to perform pre- and post processing steps of similar complexity. The hierarchical basis has the advantage that the hierarchical coefficients $\alpha_{\ell,i}$ for grid points $x_{\ell,i}$ not in a certain component grid $C_\ell$ are 0 for this component grid. Hence, these $\alpha_{\ell,i}$ neither need to be computed nor communicated and the hierarchical basis can be thought of as compressed representation of the function represented on the component grid. The nodal values $\beta_{\ell,i}$ at grid points not in the component grid, however, may take arbitrary values. So they need to be communicated and, before that, computed. This can be done by interpolation. If \textit{Sparse Grid Reduce} is applied to component grids represented in the full grid basis, the nodal values of all grid points of $\text{SG}_n^d \setminus C_\ell$ hence need to be first computed with interpolation. As $\text{SG}_n^d \setminus C_\ell$ can include many more grid points than the
component grid $C_\ell$ itself, this significantly increases the computational workload as well as the amount of data that needs to be communicated. A computational postprocessing step like dehierarchization, however, is not necessary if Sparse Grid Reduce is applied to component grids in the full grid basis.

As the level-index vector is not required to describe the communication schemes, this chapter uses $i$ and $j$ for general indexing purposes.

The rest of this chapter is organized as follows: in Section 7.2 additional notation regarding sparse grids is summarized and important observations regarding the hierarchical basis are stated. Section 7.3 introduces the communication model in detail. The communication algorithms as well as lower bounds are presented in Section 7.4. The communication costs of the communication schemes are analyzed in detail in Section 7.5. The experimental setup as well as the employed HPC systems are described in Section 7.6. Thereafter, estimations based on the communication model as well as the experimental results are presented in Section 7.7. Section 7.8 concludes this work.

research contributions. The algorithms presented in this chapter as well as the initial analysis of the algorithms are joint work with Riko Jacob, Mario Heene, Dirk Pflüger and Markus Hegland and have been presented as [HJH+14]. The adaption and detailed analysis of the algorithms, the model to predict the execution times as well as the high-dimensional experiments are joint work with Mario Heene, Riko Jacob and Dirk Pflüger and are an ongoing project [HHJP14]. This chapter contains large text parts of both manuscripts. It is not always easy to divide the work done together, in particular if the project has developed over more than two years as this one has. I am deeply thankful for the ongoing discussions with all co-authors, despite the physical distance between us. My contributions to this chapter include the development of the communication model, the analysis of the algorithms and the prediction of the execution times of the algorithms. Mario Heene’s contributions include the implementation of the algorithms as well as performing the experiments. As the model has been developed to predict the runtimes of the algorithms and the experiments validate the predictions of the model, these two parts are hard to separate. Hence, both, the predictions of the model and the experimental results, are presented in this work. It is planned that the topics of this chapter are also covered with a different focus in the PhD thesis of Mario Heene.

7.2 observations and further notation

This section formulates two observations and introduces further notation for sparse grids to prepare the description and analysis of the communication schemes. Recall from Section 4.3, that any function $f_\ell \in V_\ell$ is
7.2 Observations and Further Notation

uniquely represented by its full grid or hierarchical coefficients. All that is required for communication is the vector of its coefficients. The advantage of the hierarchical basis is that the respective coefficients, i.e., the hierarchical surpluses, of grid points that are not part of a component grid are 0 for this component grid. This observation enables to reduce the total communication volume and is exploited by Subspace Reduce. As a result, Subspace Reduce has to work in the hierarchical basis. Let us formalize this in two observations. It is essential for both observations that we are working in the hierarchical and not in the nodal full grid basis.

**Observation 7.1.** Let \( f_\ell \in V_\ell \subset V^{SG}_n \) be represented in the hierarchical sparse grid space \( V^{SG}_n = \bigoplus_{\ell' \leq \ell} W_\ell' \) as \( f_\ell(x) = \sum_{\ell' \leq \ell} w_{\ell'}(x) \) with \( w_{\ell'} \in W_{\ell'} \). For all \( \ell' \not\leq \ell \) it holds that \( w_{\ell'}(x) \equiv 0 \), i.e., all coefficients of \( f_\ell \) relating to the basis functions spanning \( \bigoplus_{\ell' \not\leq \ell} W_{\ell'} \) are 0.

For a \( V_\ell \) and a function \( f \) define the projection \( f|_{V_\ell} \) by \( f|_{V_\ell} \in V_\ell \) and \( f|_{V_\ell}(x) = f(x) \ \forall x \in C_\ell \).

**Observation 7.2.** For \( f = \sum_{\ell' \leq \ell} w_{\ell'} \in V_\ell \) and a \( V_k \) it holds that: \( f|_{V_k} = \sum_{\ell' \leq \ell} w_{\ell'} \).

By Observation 7.1, representing a function \( f_\ell \in V_\ell \) (which is given in the hierarchical basis) in the sparse grid space \( V^{SG}_n \) is achieved by filling in zeros. Projecting \( f_\ell \in V_\ell \) from one anisotropic space to another one, \( V_k \), is done by sampling the hierarchical surpluses of \( f_\ell \) at the grid points common to \( V_\ell \) and \( V_k \) according to Observation 7.2. All grid points that are in \( V_k \) but not in \( V_\ell \) again have a 0 coefficient. In the nodal grid basis, both operations, i.e. representing \( f_\ell \) in the sparse grid space \( V^{SG}_n \) and projecting \( f_\ell \) onto \( V_k \), are more complicated and require interpolation.

In the analysis of the communication schemes and the lower bounds, the set of component grids \( CG(H_\ell) \) that contributes to a particular hierarchical increment space \( H_\ell \), i.e., that contain the grid points of \( H_\ell \), is important (see also the right part of Figure 7.1):

\[
CG(H_\ell) = \{ C_{\ell'} \in C^d_n : H_\ell \subseteq C_{\ell'} \} = \{ C_{\ell'} \in C^d_n : \ell' \geq \ell \}.
\]

For a set \( A \subset C^d_n \) of component grids, we further define the set of grid points shared with the component grids in \( C^d_n \setminus A \) (see also the right part of Figure 7.1):

\[
sharedG(A) = \bigcup_{C_\ell \in A} C_\ell \cap \bigcup_{C_\ell \in (C^d_n \setminus A)} C_\ell = \bigcup_{H_\ell \in C^d_n : CG(H_\ell) \cap A \neq \emptyset, \ CG(H_\ell) \notin \mathcal{A}} H_\ell.
\]

If the component grids of \( A \) are stored on one set of nodes while all other component grids \( C^d_n \setminus A \) reside on a different set of nodes, these are the grid points that are contained on both sets of nodes and hence need to be exchanged.
The models for distributed systems discussed in Section 2.2 usually assume that the transmission of a message incurs two kinds of cost: a latency or startup term to establish the communication channel independent of the size of the message to be transmitted and a bandwidth term which depends on the size of the message sent. The models for the memory hierarchy also replicate these two costs as they only allow data exchange in blocks of predetermined size. For fixed message size the bandwidth term is constant and hence can be added to the constant latency term. We use these two basic costs in the communication model used to analyze the communication schemes of this chapter.

The communication model is designed to be simple and similar to the BSP model, the logP model and the models typically used to analyze message passing operations [TRG05, PY07]. The model is designed to capture the trade-off between reducing the total communication by splitting the communication into many small messages versus accumulating all necessary information in few big messages which is likely to increase the total communication volume. The model can be used to analyze any algorithm that deals with this trade-off. The details and assumptions of the model are: the communication nodes (in the following only nodes) are uniformly connected, i.e., the topology of the communication network is a clique, as for the BSP and logP model. The communication is synchronous and round-based as in the PEM, BSP and MapReduce model. Per round, every node can either send or receive a single message of arbitrary size per round. At the beginning of the round the communication pairs are fixed and then one message is transmitted from sender to receiver (in the BSP model this is called 1-relation). The time to communicate a message of size $m$ is determined by two parameters, the latency $L$ and the bandwidth $B$, and given by $L + \frac{m}{B}$ similarly to the BSP and logP model. The time needed for one round is the time taken to send the largest message of that round. We assume that the nodes need
to buffer incoming and outgoing messages and the node size includes this buffer besides the data already stored at the node. As the focus is on the pure communication task and the communication schemes differ by the messages that are sent, we disregard local computations as in the I/O, EM and PEM model.

We assume that there is precisely one node per component grid responsible for the communication. This work focuses on basic communication patterns and does neither address load balancing nor computing several component grids on one node, splitting a component grid onto several nodes or the combination of the latter two. As we study communication pattern, the computations on a node, e.g., summation, copying or hierarchization and dehierarchization as pre- and postprocessing, are not considered. Precisely, we study the following communication task:

**Input:** A set of nodes $N_\ell$ ($\ell$ such that $C_\ell \in \mathcal{C}_n^d$). Each node $N_\ell$ stores the values of $f_\ell$ (e.g., the solution of a PDE) at the grid points of component grid $C_\ell$.

**Output:** For all $N_\ell$: Node $N_\ell$ stores $f_{n \mid V_\ell}^{CT}$, i.e. the values of the combination technique solution $f_{n \mid V_\ell}^{CT}$ (see Equation 4.5) at the grid points of component grid $C_\ell$.

The presented algorithms are analyzed with respect to the following cost measures:

- **total number of messages sent,**

- **number of rounds performed,**

- **total communication volume,**

- **makespan communication volume** (MkVol) (maximum communication volume per round summed over all rounds),

- **maximum node size,**

- **total number of communication nodes.**

The overall communication time is given by the number of rounds times the latency $L$ plus the makespan communication volume divided by the bandwidth $B$,

$$L \cdot \text{(number of rounds)} + \frac{1}{B} \cdot \text{(makespan volume)}.$$ (7.1)
<table>
<thead>
<tr>
<th>Arbitrary Dimension</th>
<th>Lower Bound</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages sent:</td>
<td>$2 \cdot (</td>
<td>e^d_n</td>
<td>- 1)$</td>
</tr>
<tr>
<td>Number of rounds:</td>
<td>$2 \cdot (\log_2</td>
<td>e^d_n</td>
<td>)$</td>
</tr>
<tr>
<td>Total volume:</td>
<td>$2 \cdot \left( \sum_{H \in \mathcal{C} \mathcal{G}^d_{n-1}}</td>
<td>H</td>
<td>(\mathcal{G}(H) - 1) \right)$</td>
</tr>
<tr>
<td>Makespan volume:</td>
<td>$2 \cdot \left( \max_{C \subseteq e^d_n}</td>
<td>C</td>
<td>\right)$</td>
</tr>
<tr>
<td>Maximum node size:</td>
<td>$\max_{C \subseteq e^d_n}</td>
<td>C</td>
<td>\cdot</td>
</tr>
<tr>
<td>Communication nodes:</td>
<td>$</td>
<td>e^d_n</td>
<td>$</td>
</tr>
</tbody>
</table>

Table 7.1: Complexity bounds for the communication task of the sparse grid combination technique for dimensions $d$ and level $n$. Ceiling functions are omitted for a simpler presentation.
This section describes two general algorithmic approaches, Sparse Grid Reduce and Subspace Reduce, as well as lower bounds for the communication task of the combination technique for a regular sparse grid of dimension $d$ and level $n$ without boundary. The only communication operations the communication schemes are going to perform are AllReduce-operations. Hence, the performance of the algorithms depends on the performance of the AllReduce implementation. For the analysis carried out in this chapter, we assume that the communication task, and hence also the AllReduce-task, is split into two distinct phases, reduce and broadcast. The reduce step assembles the values of the combination technique solution $f_{CT}^n$ as given by Equation 4.5, for all grid points $x \in \text{SG}_n^d$. It is allowed that the combination technique solution is distributed over several compute nodes. The broadcast phases distributes the combination technique solution $f_{CT}^n$ to all compute nodes. This splitting into the two distinct phases, reduce and broadcast, enables us to proof that each phase is solved optimally with respect to the total number of rounds, the number of messages sent and the total communication volume by binomial trees.

While binomial trees are a good solution with respect to the cost measures they optimize for distinct reduce and broadcast steps, there other efficient solutions for the AllReduce-task. In practice, different versions of AllReduce might be used depending on the size of the set that AllReduce is applied to. If the analysis carried out in this chapter would take into account that AllReduce is implemented differently depending on the size of the set to be reduced, the complexity of the results would increase without bringing any further insights. Hence, we furthermore assume for the analysis of the communication schemes that binomial trees are used to implement the two distinct phases of the AllReduce-task. Different implementations of AllReduce and their effects on the presented communication schemes are discussed in Section 7.4.1. The assumption of separate reduce and broadcast as well as the assumption that both phases are implemented using binomial trees is not essential for the optimality of the communication schemes. Assume that AllReduce is implemented such that it minimizes one of the cost measures for the AllReduce task (while reduce and broadcast can be merged). Furthermore, assume that a communication scheme is optimal with respect to the same cost measure under the assumption of separate reduce and broadcast (and that binomial trees are used to implement both phases). Then, the communication scheme is also optimal with respect to this cost measure when reduce and broadcast can be merged (both, reduce and broadcast, are then likely to use schemes different from binomial trees for communication).
The two presented communication schemes are depicted in Figure 7.1. The first communication scheme, *Sparse Grid Reduce*, expands each component grid to the whole sparse grid before performing a single *AllReduce*-operation for the whole sparse grid and all compute nodes. Given the assumptions that the reduce and broadcast step are distinct (and performed using binomial trees), *Sparse Grid Reduce* thus minimizes the number of messages sent and the number of rounds performed at the expense of a larger communication volume.

The second communication scheme, *Subspace Reduce*, takes advantage of the decomposition of a sparse grid into its hierarchical increment spaces. Only component grids that share a certain increment space exchange the values of this increment space using an *AllReduce*. For *Subspace Reduce* it is crucial that the values of the component grids are hierarchized, i.e., they are given in the hierarchical basis instead of the full grid basis. Under the assumptions that the reduce and broadcast step are distinct (and performed using binomial trees), *Subspace Reduce* minimizes the total communication volume while the number of messages and rounds increases.

This section continues with the derivation of the lower bounds and the complexities of both approaches for the different cost measures. The complexities are summarized abstractly in Table 7.1. For the lower bounds we assume that an *AllReduce*-operation is split into a reduce and a broadcast step. For the upper bounds we furthermore assume that each of these steps is implemented using binomial trees. *Subspace Reduce* is then adapted to allow for parallelism. Furthermore, both approaches are then generalized to sparse grids with boundary points and sparse grids including a minimum level. Generalizations to other types of sparse grids are straightforward as long as the adaptivity always includes whole increment spaces. We do always assume that the component grids store the hierarchical surpluses and not the original function values as this enables us to take advantage of the two observations.

7.4.1 *Analysis and Discussion of* *AllReduce*

The *AllReduce*-operation solves the following task: assume m communication nodes want to combine a vector v. Initially, each node i holds its version v_i of the vector, and after the execution of *AllReduce* it wants to store the sum \( \sum_{i=1}^{m} v_i \) on every node.

We split this task into a reduce phase that creates the sum and a broadcast phase that distributes it. We claim that each of the two phases can be solved optimally with respect to the number of rounds, the number of messages sent and the total communication volume by binomial trees in the following way: arrange the communication nodes in form of a binomial tree of height \( h = \lceil \log_2 m \rceil \) and consider the reduce phase first. In one round every leaf sends its vector v_i of partial sums to its
parent node who adds it to its own vector. Then the current leaves are
deleted and the next round starts. For \( m < 2^h \) the binomial tree may
be incomplete such that up to two leaves want to send their vector to
the same parent. In that case, one leaf delays its message until the next
round. When only one node is left, the combined vector
\( v_{\text{res}} := \sum_{i=1}^{m} v_i \)
is stored at the root and the reduce phase is completed. In total \( \lceil \log_2 m \rceil \)
rounds are sufficient and the makespan volume is \( |v| \cdot \lceil \log_2 m \rceil \) for the
reduce phase.

In the broadcast phase the messages are sent in reverse order copying
the combined vector \( v_{\text{res}} \) to all nodes in \( \lceil \log_2 m \rceil \) rounds. For both
phases together

\[
\text{the number of rounds is } 2 \cdot \lceil \log_2 m \rceil \\
\text{and the makespan volume is } 2 \cdot |v| \cdot \lceil \log_2 m \rceil.
\]

Observe that the communication of the reduce and broadcast phases
are time inverse to each other, with the local operation copy (broadcast)
instead of summing (reduce).

To see that binomial trees solve both the reduce and the broadcast step
optimal with respect to the number of rounds, the number of messages
sent and the total communication volume note that: at least \( m - 1 \)
messages, one less than the number of participating nodes, need to be sent
for either the reduce or the broadcast task. Also, each entry of \( v \) must
be sent at least \((m - 1)\) times. Hence, the total volume for either reduce
or broadcast is \( |v| \cdot (m - 1) \). As the number of nodes which have yet pro-
vided their information in the reduce step (received their information in
the broadcast step) can at most be doubled per round, a lower bound for
the number of rounds for either step is \( \lceil \log_2 m \rceil \).

Throughout this chapter we assume that \textit{AllReduce} is split into the
these two distinct phases, reduce and broadcast. For the analysis of
the communication schemes we assume, in addition, that \textit{AllReduce}
is implemented using binomial trees. In fact many common implementa-
tions of \textit{AllReduce} use binomial trees. In practice, also implementations
of \textit{AllReduce} that merge both phases and do not use binomial trees are
employed. Also, the model [TRG05, PY07] typically used to analyze
message-passing operations like \textit{AllReduce} assumes that communication
links are bidirectional, i.e. a node can send and receive a message simul-
taneously. Hence, this model differs from the model of Section 7.3.

Given bidirectional communication links, \textit{AllReduce} can merge the
reduce and the broadcast phase by means of a recursive-doubling ap-
proach [TRG05]. Such an algorithm reduces the number of rounds as
well as the makespan volume to that of a single phase, i.e. the number
of rounds is \( \lceil \log_2 m \rceil \) and the makespan volume is \( |v| \cdot \lceil \log_2 m \rceil \). As the
lower bound of \( \log_2 m \) rounds for one phase, either reduce or broadcast,
carries over to the \textit{AllReduce}-task, an implementation of \textit{AllReduce} based
on a recursive doubling approach works in the minimum number of
rounds. The total communication volume and the total number of messages increases, however, as each node communicates the whole vector in each round.

Furthermore, AllReduce implementations that reduce the makespan volume below the makespan volume communicated by binomial trees exist and are used in practice if the set to be reduced is large. The combination of a reduce-scatter with an allgather operation yields such an algorithm [TRG05]. Both operations can be implemented by passing chunks of size $|v|/m$ between the processors in a round-robin fashion in $m - 1$ rounds. Such an implementation of AllReduce communicates the minimum total volume [PY07]. The optimal makespan volume comes at the price of an increased number of rounds and messages sent.\footnote{The reduce-scatter and the allgather operation can also be implemented by recursive halving or recursive doubling [TRG05] but because of the two distinct phases the number of rounds is still twice that of the lower bound.}

As the only communication operations performed by the communication schemes are AllReduce-operations, the optimality of the communication schemes does not rely on the assumption of separate reduce and broadcast or that binomial trees are used for each phase. In contrast, the communication schemes inherit optimality with respect to a certain cost measure from the AllReduce-operation, if they were optimal with respect to that cost measure for separate reduce and broadcast and using binomial trees for both phases. In detail: if AllReduce is implemented such that it minimizes the number of rounds for the AllReduce-task, then Sparse Grid Reduce is also executed in the minimum number of rounds with respect to the communication task of the combination technique. Similarly, if AllReduce is implemented such that it minimizes the total communication volume, then also Subspace Reduce is executed communicating the minimum total volume.

7.4.2 Algorithmic Notation

For the brief algorithmic description of the algorithms, we need the following functions:

- **Buffer S**: copy the values of the set $S$ of coefficients into the send buffer.

- **AllReduce($S$, $\mathcal{C}$)**: perform an AllReduce operation for the set $S$ using the communication nodes $\mathcal{C}$. In more detail: if $S_j$ is the copy of $S$ stored initially on $C_j \in \mathcal{C}$ and the values in $S_j$ can be indexed by $s_{j,i}, i \in I$, then each value of $S$ is reduced in the sense that $s_{\text{res}, i} = \sum_{j, C_j \in \mathcal{C}} s_{j,i}$ and the resulting set $S_{\text{res}} = \bigcup_{i \in I} s_{\text{res}, i}$ is stored at all nodes $C_j \in \mathcal{C}$.

- **Extract $S$ from Buffer**: Copy the values of the set $S$ from the buffer to the corresponding positions of the local copy of $S$. 

7.4.3 Lower Bounds

The lower bounds for the different cost measures are summarized in Table 7.1. For a sparse grid of level $n$ and dimension $d$ the communication task consists of $|e^d_n|$ component grids and hence this is the minimum number of communication nodes required. Also, the communication nodes have to be at least as large as the largest component grid, $\max_{C_{\ell} \in e^d_n} |C_{\ell}|$. For all other cost measures we only analyze the reduce phase as the communication requirements of the broadcast phase are identical. As the root increment space $H_1$ is part of all $C_{\ell} \in e^d_n$ we need at least $|e^d_n| - 1$ messages to calculate the combination technique solution at the grid point of this increment space. Furthermore, as the number of nodes over which the overall sum for $H_1$ is currently still split can at best be halved in each round, there are at least $\lceil \log_2 |e^d_n| \rceil$ communication rounds. To lower bound the total communication volume, note that each point of the sparse grid needs to be communicated at least one time less than it is element of a component grid, namely,

$$\sum_{H_{\ell} \in \mathcal{H}_n^d} |H_{\ell}| \cdot (\text{CG}(H_{\ell}) - 1) = \left( \sum_{C_{\ell} \in e^d_n} |C_{\ell}| \right) - |SG^d_n|.$$

A trivial lower bound for the makespan communication volume is

$$\max_{C_{\ell} \subset e^d_n} |\text{sharedG}(C_{\ell})|.$$

7.4.4 Sparse Grid Reduce (Algorithm 7.1) – Assembling the Sparse Grid

Sparse Grid Reduce is a structural simple approach serving as baseline. Fix an order of the hierarchical increment spaces of the sparse grid, e.g. the lexicographic order of the level vectors. Each component grid then creates a copy of the sparse grid $SG^d_{n-1}$ in the send buffer by copying the hierarchical increment spaces contained in the component grid and filling in zeros for the hierarchical increment spaces not present in the component grid but in the sparse grid $SG^d_{n-1}$. Now, the communication task is a single AllReduce on all hierarchical increment spaces that are part of $SG^d_{n-1}$. See Algorithm 7.1 for the algorithmic description. Expanding the component grids to the sparse grid of level $n - 1$ (and not $n$) is enough as increment spaces of level sum $n$ are only part of a single component grid and do not need to be communicated.

The complexities of Sparse Grid Reduce with respect to the different cost measures are summarized in Table 7.1. Sparse Grid Reduce sends the minimum number of messages and uses the minimum number of rounds and computation nodes. The total as well as the makespan communication volume, however, increase significantly. Furthermore, every
node has to be large enough to store a sparse grid of level \( n - 1 \) (and additionally a single, local component grid of level sum \( n \)) which may not be possible for large levels and dimensions.

Sparse Grid Reduce does not rely on communicating the hierarchical surpluses but could also send the function values sampled at the sparse grid points. This changes the preprocessing phase from hierarchization to interpolation to all points in \( \text{SG}^d_{n-1} \) which are not in the component grid of the respective node.

7.4.5 Subspace Reduce (Algorithm 7.2) – Communicating Each Hierarchical Increment Space Separately

Subspace Reduce is the other extreme splitting the communication task into subtasks determined by the hierarchical increment spaces as suggested by Observation 7.2. First, fix an order of the hierarchical increment spaces. Then, for all hierarchical increment spaces \( H_\ell \in H^d_{n-1} \) perform an AllReduce employing all component grids containing \( H_\ell \), i.e. the nodes \( \text{CG}(H_\ell) \). Reducing hierarchical increment spaces of level sum \( n \) is not necessary as those belong only to a single component grid. The algorithmic description of Subspace Reduce is given by Algorithm 7.2.
The complexities of \textit{Subspace Reduce} with respect to the different cost measures are summarized in Table 7.1. As only nodes that require a particular $H_{\ell}$ are participating in the communication subtasks, this approach achieves minimum total volume. As a drawback, splitting the communication according to hierarchical increment spaces significantly increases the number of messages sent. The makespan communication volume highly depends on the possibility to reduce different hierarchical increment spaces in parallel. For the analysis of (the original version of) \textit{Subspace Reduce} we make the pessimistic assumption that all increment spaces are reduced in serial. In fact, when the increment spaces are arranged in the naïve lexicographic order of their level vector, they almost need to be reduced in serial. In the lexicographic order, the set of component grids for increment spaces that are reduced in successions almost always overlap (except when the maximum level is reached in one dimension and a jump in the lexicographic order is performed), and hence \textit{Subspace Reduce} has to reduce the increment spaces almost in serial. Section 7.4.6 introduces a simple order of the hierarchical increment spaces that allows for the parallel execution of the \textit{AllReduce}-operations.

\textit{Subspace Reduce} relies on working on the hierarchical surpluses. By Observation 7.1, the hierarchical surplus of a grid point not in a certain component grid $C_{\ell}$ is 0 for this component grid. This enables us to limit the reduce procedure for an increment space $H_{\ell}$ to the subset $C(H_{\ell})$ of component grids. While the hierarchical surpluses are 0, the function values at grid points not in the component grid may be arbitrary and hence working in the nodal basis is not possible when employing \textit{Subspace Reduce}.

7.4.6 Enabling Parallelism: the Order of Increment Spaces for Parallel Subspace Reduce

\textit{Subspace Reduce} (Algorithm 7.2) does not specify the order in which the increment spaces are reduced and the analysis of the original \textit{Subspace Reduce} method assumes that all increment spaces are reduced in serial. When increment spaces belong to disjoint sets of component grids, however, they can be reduced in parallel. This section gives a simple order of the increment spaces that enables parallelism of the \textit{AllReduce}-operations.

Denote by $a \in \{0, \ldots, n-1\}$ the distance from the sparse grid level $n$ and consider all increment spaces of fixed level sum $(n + d - 1) - a$. For a fixed level sum, the increment spaces of this level sum have $d - 1$ degrees of freedom remaining. We can project the level vector $\ell$ of all increment spaces of fixed level sum into a $d - 1$ dimensional space by projecting along dimension $d$, i.e., by ignoring the $d$th component of the level vector. As we have fixed the level sum, the level vector of the increment spaces has to differ in the first $d - 1$ components and hence all
Figure 7.2: **Left:** reducing increment spaces in parallel for \( a = 1 \). The increment spaces corresponding to the shaded component grids can be reduced in parallel requiring the framed component grids. In 2 dimensions the \( d - 1 \) dimension hypercubes correspond to intervals (dashed). Increment spaces within the same interval cannot be reduced in parallel as they share component grids. **Right:** adjusting increment spaces when introducing a minimum level (here \( \ell_{\text{min}} = (2, 3) \)). Increment spaces that are in the same box are merged. Increment spaces which do no longer take part in a component grid are removed. Increment spaces whose level \( \ell \) suffices \( \ell \geq \ell_{\text{min}} + 1 \) remain unchanged.

increment spaces of a fixed level sum are projected to different positions. Also, for a fixed level sum, there is a 1–1 (bijection) correspondence between the projected level vector \( \ell_{(d-1)} \) and the original level vector \( \ell \). The projection to \( d - 1 \) degrees of freedom has the purpose to navigate easily on all component grids of a fixed level sum as we are going to handle only increment spaces of the same level sum in parallel. Denote by \( e_r \) the unit vector in dimension \( r \). Any pair of increment spaces of level sum \( (n + d - 1) - a \) whose level vector is at stride \( (a + 1) \cdot e_r \) for some \( r \in \{1, \ldots, d - 1\} \) is not contained in the same component grid and can be reduced in parallel (see also Figure 7.2). Alternatively to an access at a certain stride, one can also think of partitioning the increment spaces of a fixed level-sum into \( d - 1 \) dimensional hypercubes of side length \( a \). All increment spaces at the same relative position within their hypercube can be reduced in parallel.

**Observation 7.3.** Fix a distance \( a \in \{0, \ldots, n - 1\} \) from the maximum refinement level \( n + d - 1 \) and one level vector \( \ell \) at that distance \( a \), i.e. \( \|\ell\|_1 = (n + d - 1) - a \). Out of all increment spaces \( H_{\ell'} \) with \( \|\ell'\|_1 = (n + d - 1) - a \), those whose projected level vector \( \ell'_{(d-1)} \) suffices

\[
\ell'_{(d-1)} = \ell_{(d-1)} + (a + 1) \cdot \sum_{r=1}^{d-1} c_r \cdot e_r 
\]

for \( c_r \in \mathbb{Z} \) can be reduced in parallel.
Proof. All increment spaces for \( a = 0 \) can be handled in parallel as each one only concerns a single component grid. In fact, these increment spaces do not need to be reduced at all as they only belong to a single component grid.

To reduce increment spaces of level sum \( n + d - 2 \), i.e. \( a = 1 \), we need the component grid \( C_\ell \) and also the component grids that are refined once further than \( \ell \) in every dimension. Hence, if we reduce \( H_\ell \), then \( C_\ell \) and all component grids \( C_{\ell'} \) with \( \ell' = \ell + e_r \) for any \( 1 \leq r \leq d \) are participating in the current reduce. This means, all \( H_{\ell'} \) with \( \ell'_{(d-1)} = \ell_{(d-1)} + 2 \sum_{r=1}^{d-1} c_r \cdot e_r \) for \( c_r \in \mathbb{Z} \) can be reduced in parallel.

To reduce increment spaces of level sum \( n + d - 3 \), i.e. \( a = 2 \), we need \( C_\ell \) and also at most the component grids that are twice further refined than \( \ell \). Hence, if we reduce \( H_\ell \) then the component grids \( C_{\ell'} \) with \( \ell' = \ell + \sum_{r=1}^{d} b_r \cdot e_r \) for \( b_r \in \{0, 1, 2\} \) are a super set (we are overestimating for \( a \geq 2 \) as not all of these component grids exist) for the component grids needed for this reduce. Therefore we can reduce all \( H_{\ell'} \) for \( \ell'_{(d-1)} = \ell_{(d-1)} + 3 \sum_{r=1}^{d-1} c_r \cdot e_r \) for \( c_r \in \mathbb{Z} \) in parallel.

For general \( a \), we need at most the component grids that are \( a \) times further refined than \( \ell \). Hence, if we reduce \( H_\ell \) then the component grids \( C_{\ell'} \) with \( \ell' = \ell + \sum_{r=1}^{d} b_r \cdot e_r \) for \( b_r \in \{0, 1, 2, \ldots, a\} \) are a super set for the component grids needed for this reduce. Therefore we can reduce all \( H_{\ell'} \) for \( \ell'_{(d-1)} = \ell_{(d-1)} + (a + 1) \cdot \sum_{r=1}^{d-1} c_r \cdot e_r \) for \( c_r \in \mathbb{Z} \) in parallel.

Above, we have assumed that all the component grids of the given level vectors exist. That is usually not the case as component grids only exist for level sums between \( n \) and \( n + d - 1 \). Hence, Observation 7.3 does not state all parallelism that can be exploited for the \textit{AllReduce}-operations as the given superset for the component grids necessary for a reduce step is slightly pessimistic.

The parallelism described by Observation 7.3 can be exploited by 

\textbf{Subspace Reduce (Algorithm 7.2)}

when the hierarchical increment spaces \( H_\ell \) are ordered by:

- level sum \( \|\ell\|_1 \) increasing from \( d \) to \( n + d - 1 \) (distance \( a \) decreasing from \( n - 1 \) to 0),
- relative position of \( H_\ell \) within its \((d-1)\)-dimensional hypercube of side length \( a \),
- \((d-1)\)-dimensional hypercubes of side length \( a \) that partition the increment spaces of level sum \( \|\ell\|_1 = (n + d - 1) - a \).

The given order of the increment spaces is not an optimal parallelization method but a simple one already allowing for a lot of parallelism, at least for small \( a \). For small \( a \) (for a high level sum) there are a lot of hierarchical increment spaces and those increment spaces are large. For
these two reasons parallelization is most important for small \( a \). With increasing \( a \) the parallelism introduced by the given order decreases as the tiling hypercubes grow. Also, for larger \( a \) the number of hierarchical increment spaces of level sum \( (n + d - 1) - a \) decreases as well as their size. Hence for larger \( a \) parallelizing should not be as important. In general, the parallelization potential of every order is limited for larger \( a \) as the number of component grids containing an increment space grows as \( a \) increases.

The employed access at stride is simple with respect to the index computations needed to select the increment spaces that can be reduced in parallel. As we have seen that this simple approach already allows for a lot of parallelism for small \( a \), other parallelization approaches promise only minor improvements and are thus not considered in this work.

### 7.4.7 Adapting the Algorithms to Treat Generalizations of Sparse Grids

This section describes the necessary modifications such that the two algorithms \textit{Sparse Grid Reduce} and \textit{Subspace Reduce} can handle sparse grids with boundary, with minimum level or both.

#### 7.4.7.1 Boundary Treatment

There are three main approaches to equip sparse grids with boundaries as discussed in Section 4.4. The existing functions can be modified to extrapolate towards the boundary, new basis functions can be added to existing increment spaces (level 1 boundary) or new increment spaces can be created to host the new basis functions (level 0 boundary). See Figure 4.3 for the respective modified 1-dimensional basis functions. While the experiments focus on the second case in which boundary functions are added to existing hierarchical increment spaces (level 1 boundary) this section describes the necessary modifications of the algorithms for all three types of boundary functions.

For basis functions extrapolating towards the boundary, no modifications to the algorithms are necessary as no new grid points are introduced.

When using a level 1 boundary, new boundary functions are added to existing hierarchical increment spaces. Hence, the set of hierarchical increment spaces is not altered for this kind of boundary. As a result, the definitions of both algorithms, \textit{Sparse Grid Reduce} and \textit{Subspace Reduce}, still apply. Both algorithms work as before in the sense that still the same nodes need to communicate. We say: the structure of the communication scheme does not change. Only the size of the messages changes.

When using a level 0 boundary, new hierarchical increment spaces with \( \min(\ell) = 0 \) are created to host the new boundary basis functions. This, in turn, also adds additional component grids and changes the set of hierarchical increment spaces \( \mathcal{H}^d_{n-1} \) and the set of component
7.4 Algorithms and Lower Bounds

7.4.7.2 Introducing and Handling a Minimum Level

A minimum level $\ell_{\min}$ is a threshold for the refinement level of the component grids excluding component grids from the combination that do not meet this minimum refinement criteria. There are two practical considerations that lead us to introducing a minimum level. First, experiments [KPJH12] conducted with GENE [GLB+11] showed that highly anisotropic grids should be excluded from the combination as a minimum discretization is required to represent physical properties. Second, the minimum level enabled us to run the numerical experiments of the next section. Increasing the level $n$ of the sparse grid increases the number of component grids $\left| C^d_{dn} \right|$ in the combination. As we are assuming that each component grid is stored on its own node, this requires us to run the experiments on at least $\left| C^d_{dn} \right|$ nodes. To give a couple of figures, for $n = 10$ there are 136 component grids for $d = 3$, there are 1,876 component grids for $d = 5$ and there are 92,378 component grids for $d = 10$ assuming no boundary or boundary of level-1 type. As we had a limitation on the number of nodes available for our experiments, the minimum level enables us to limit the number of component grids while still obtaining representative scenarios. If the level sum of the minimum level $\| \ell_{\min} \|_1$ is increased by the same amount as the refinement level $n$ of the sparse grid, the number of component grids of the combination stays constant. As an example, for $d = 5$ there are 456 component grids for $n = 7$ and $\ell_{\min} = (1, 1, 1, 1, 1)$ as well as for $n = 8$ and $\ell_{\min} = (1, 1, 1, 1, 2)$ and $n = 17$ and $\ell_{\min} = (3, 3, 3, 3, 3)$. Furthermore, increasing the minimum level by the same amount as the sparse grid level $n$ has the effect that the communication task keeps same structure: the same nodes need to exchange messages, only the size of the messages differs. We now first introduce the minimum level when the sparse grid contains no boundary and then generalize to sparse grid with boundary points.

Formally, let $\ell_{\min}$ denote the minimum level. Then, the altered set of component grids $\mathcal{C}^d_{(n, \ell_{\min})}$ is

$$\mathcal{C}^d_{(n, \ell_{\min})} = \left\{ C_\ell \in \mathcal{C}^d_n : \ell \geq \ell_{\min} \right\}.$$  

The component grids $C_\ell$ themselves remain unchanged. For the hierarchical increment spaces, we face two issues, however. First, some increment spaces drop out as the component grids that contained them are no longer in $\mathcal{C}^d_n$. Second, to reduce the number of messages passed by Subspace Reduce we want to merge increment spaces $H_\ell$ that belong to
the same set of component grids $CG(H_\ell)$ due to the new minimum level restriction. Figure 7.2 (right) depicts both effects.

To reduce the number of messages passed by Subspace Reduce while not increasing the communication volume, we need to merge those increment spaces that belong to the same set of component grids. Formally, for a level vector $\ell$ define the set

$$I(\ell, \ell_{\min}) = \{ r \in \{1, \ldots, d\} : \ell_r = (\ell_{\min})_r \}$$

stating which dimensions are refined minimally. With that on hand we can create sets $K(\ell, \ell_{\min})$ of level vectors whose corresponding increment spaces are going to be merged:

$$K(\ell, \ell_{\min}) = \left\{ \ell' : (1 \leq \ell'_r \leq \ell_r \text{ for } r \in I(\ell, \ell_{\min}) \land (\ell'_s = \ell_s \text{ for } s \not\in I) \right\}.$$

The sets $K(\ell, \ell_{\min})$ can be used to assemble the merged increment spaces $H(\ell, \ell_{\min})$ which merge the increment spaces $H_{\ell'}$ with a level $\ell'$ below the minimum level with the increment space $H_{\ell}$ of smallest level $\ell$ which is at the threshold $\ell_{\min}$ and larger than $\ell'$,

$$H(\ell, \ell_{\min}) = \bigcup_{\ell' \in K(\ell, \ell_{\min})} H_{\ell'}.$$

The hierarchical increment spaces $H_\ell$ for which $I_\ell \neq \emptyset$ are changed, all others stay as before. Now let us consider the set of hierarchical increment spaces and exclude those no longer relevant for the combination. The set $H^d_{\ell}(n, \ell_{\min})$ taking the minimum level $\ell_{\min}$ into account is

$$H^d_{\ell}(n, \ell_{\min}) = \left\{ H(\ell, \ell_{\min}) : \ell \text{ such that } H_\ell \in H^d_{n} \text{ and } \ell \geq \ell_{\min} \right\}.$$

In contrast to the case without minimum level, certain increment spaces are excluded and grid points are removed from the sparse grid such that the sparse grid with minimum level $\ell_{\min}$ is given by

$$SG^d_{\ell}(n, \ell_{\min}) = \bigcup_{H(\ell, \ell_{\min}) \in H^d_{\ell}(n, \ell_{\min})} H(\ell, \ell_{\min}).$$

With the merged increment spaces we can keep the number of messages passed by Subspace Reduce to two (one for the reduce and one for the broadcast step) for each set of grid points that is contained in a different set of component grids. For both, Sparse Grid Reduce and Subspace Reduce, simply excluding the hierarchical increment spaces no longer needed for the combination results in correct algorithms.

Note that the minimum level can be introduced in the same way when the sparse grid contains boundary points. It does not matter which of the three types of boundaries discussed in Section 7.4.7.1 is employed. In fact, a minimum level and boundary basis functions of type 1 affect the
communication schemes in a similar manner. For a minimum level in dimension \( r \) we merge all 1-dimensional increment spaces in dimension \( r \) up to \( (\ell_{\min})_r \) to one increment space. The boundary basis functions of type 1 have also been merged with the root basis function on level 1.

7.5 **runtime analysis**

This section elaborates on the generic runtime formulas for *Sparse Grid Reduce* and *Subspace Reduce* given in Table 7.1 and derives formulas only involving the dimension \( d \) and level \( n \) of the sparse grid. We only derive the formulas for sparse grids without boundary points and no minimum level in this section and restrict the discussion to the original version of *Subspace Reduce*. It results in similar formulas if boundary points or a minimum level is included and/or *Parallel Subspace Reduce* results is used. For the experimental part in Section 7.7, model calculations are also carried out for sparse grids with boundary and/or minimum level and *Parallel Subspace Reduce*.

In general we assume that the communication time of an algorithm is given by the model discussed in Section 7.3: The number of rounds times the latency of the communication network plus the makespan volume divided by the bandwidth gives the communication time as in Equation 7.1. All the formulas build on the assumption (Section 7.4.1) that the *AllReduce*-routine is implemented using two sweeps over a binomial tree which consists of the involved communication nodes.

In the following, we first give precise but involved formulas for the communication time of *Sparse Grid Reduce* and *Subspace Reduce* in the communication model. We then simplify these bounds and derive upper bounds for the communication times.

7.5.1 **Precise Communication-Time Formulas for Sparse Grids Without Boundary and Without Minimum Level**

Denote by \( A^d_k \) the number of component grids or increment spaces of level sum \( k + d - 1 \), i.e., the number of component grids or increment spaces that are in \( C^d_k \backslash C^d_{k-1} \) and \( H^d_k \backslash H^d_{k-1} \) respectively. Observe that for a certain level sum there is the same number of component grids as of increment spaces. Using a combinatorial approach \( A^d_k \) is given by

\[
A^d_k = \binom{k + d - 2}{d - 1}.
\]
The size of a sparse grid without boundary and minimum level is given by [BG04] as
\[
\left| \text{SG}_n^d \right| = \sum_{i=0}^{n-1} 2^i \cdot \binom{d-1+i}{d-1} = 2^n \cdot \left( \frac{n^{d-1}}{(d-1)!} + \Theta(n^{d-2}) \right). \tag{7.2}
\]

As is standard in the sparse grid literature, we assume that the dimension \(d\) is constant and hence omit this constant in the \(\Theta\)-notation. Furthermore, we assume that the latency \(L\) is given in seconds per message, the bandwidth \(B\) in bytes per second and double precision (8 bytes per double) is used.

Then, the runtime for **Sparse Grid Reduce**, assuming \(n \geq d\), is
\[
\text{SG Reduce}(n, d) = 2 \cdot L \cdot \left\lfloor \log_2 \left| c_n^d \right| \right\rfloor + 2 \cdot \frac{8}{B} \cdot \left\lfloor \log_2 \left| c_n^d \right| \right\rfloor \cdot |\text{SG}_{n-1}^d| =
\]
\[
= 2 \cdot \left\lfloor \log_2 \left| c_n^d \right| \right\rfloor \left( L + \frac{8}{B} \cdot |\text{SG}_{n-1}^d| \right) =
\]
\[
= 2 \cdot \left\lfloor \log_2 \left( \sum_{i=n-d+1}^{n} A_i^d \right) \right\rfloor \cdot \left( L + \frac{8}{B} \cdot |\text{SG}_{n-1}^d| \right) =
\]
\[
= 2 \cdot \left\lfloor \log_2 \left( \sum_{i=n-d+1}^{n} \binom{i+d-2}{d-1} \right) \right\rfloor \cdot \left( L + \frac{8}{B} \cdot \sum_{i=0}^{n-2} 2^i \binom{d-1+i}{d-1} \right). \tag{7.3}
\]

For \(n < d\) the first sum has to start at \(i = 1\) instead of \(i = n - d + 1\).

The runtime for **Subspace Reduce** is
\[
\text{Subspace Reduce}(n, d) = 2 \cdot L \cdot \sum_{H \in \mathcal{G}_{n-1}^d} \left\lfloor \log_2 |C(H)| \right\rfloor + 2 \cdot \frac{8}{B} \cdot \sum_{H \in \mathcal{G}_{n-1}^d} |H| \cdot \left\lfloor \log_2 |C(H)| \right\rfloor =
\]
\[
= 2 \cdot \sum_{H \in \mathcal{G}_{n-1}^d} \left( L + \frac{8}{B} |H| \right) \cdot \left\lfloor \log_2 |C(H)| \right\rfloor =
\]
\[
= 2 \cdot \sum_{m=1}^{n-1} \left( L + \frac{8}{B} |H_\ell| \right) \cdot \sum_{\|\ell\|_1 = m + d - 1} \left\lfloor \log_2 |C(G(H_\ell))| \right\rfloor =
\]
\[
7.5 \text{ runtime analysis}
\]

\[
= 2 \cdot \sum_{m=1}^{n-1} \left[ \left( \frac{L + \frac{8}{B} \cdot 2^{m-1}}{\sum_{i=1}^{\min(d, n-m+1)} A_{m}^{n-(m-1)-(i-1)}} \right) \right] = 2 \cdot \sum_{m=1}^{n-1} \left[ \left( \frac{L + \frac{8}{B} \cdot 2^{m-1}}{\sum_{i=1}^{\min(d, n-m+1)}} \left( \frac{n + d - m - i}{d-1} \right) \right) \right]. \tag{7.4}
\]

7.5.2 Simplifying the Formulas

To get a rough idea of the complexities of \textit{Sparse Grid Reduce} and \textit{Subspace Reduce} we now simplify and upper bound the exact formulas for their communication time. The formulas are going to stay involved but provide a glimpse at the complexity of the algorithms. The simplifications are canonical and hence only the most important inequalities employed and the results are stated. In particular we use the standard estimate for binomial coefficients,

\[
\binom{k}{d} \leq \frac{k^d}{d!}. \tag{7.5}
\]

Using Equation 7.3 as starting point (assuming \( n \geq d \)), the runtime of \textit{Sparse Grid Reduce} is bounded by

\[
\text{SG Reduce}(n, d) \in \left( \frac{L + \frac{8}{B} \cdot 2^{n-1}}{\sum_{i=1}^{\min(d, n-2)}} \left( \frac{n^{d-1}}{(d-1)!} + o \left( n^{d-2} \right) \right) \right). \tag{7.6}
\]

To estimate the runtime of \textit{Subspace Reduce} we need two bounds,

\[
\sum_{m=1}^{n-1} (m + d - 2)^{d-1} \leq \int_{1}^{n} (m + d - 2)^{d-1} \, dm = \left. \frac{(m + d - 2)^{d}}{d} \right|_{m=1}^{n} \leq \left( \frac{n + d - 2}{d} \right)^d \tag{7.7}
\]
and
\[
\sum_{m=1}^{n-1} (m + d - 2)^{d-1} \cdot 2^{m-1} \leq \int_1^n (m + d - 2)^{d-1} \cdot 2^{m-1} \, dm \leq (n + d - 2)^{d-1} \cdot 2^{n-1}. \tag{7.8}
\]

The second estimate is very rough and is used in estimating the bandwidth term of Subspace Reduce which is hence overestimated.

Using Equation 7.4 as starting point, the runtime of Subspace Reduce can be bounded by

\[
\text{Subspace Reduce}(n, d) \leq \left(\frac{\log_2 (n + d)}{d-2}! \cdot \frac{(n + d)^{d-1}}{d} \cdot \left( L \cdot \frac{n + d}{d} + \frac{8}{B} \cdot 2^{n-1} \right) \right). \tag{7.9}
\]

These upper bounds for the runtime of Sparse Grid Reduce and Subspace Reduce can be used to get a rough idea which algorithm needs more rounds and which one has a larger makespan communication volume. As only upper bounds have been derived and rough estimates have been used to obtain these bounds, this is only meant as a rough indication which algorithm performs better with respect to a certain cost measure.

Comparing the number of rounds estimated for Sparse Grid Reduce (Equation 7.6) and Subspace Reduce (Equation 7.9) yields

\[
\frac{\text{Number of rounds of Sparse Grid Reduce}(n, d)}{\text{Number of rounds of Subspace Reduce}(n, d)} \approx \frac{2 \cdot d \cdot \log_2 (n + d)}{2 \cdot \log_2 (n + d)} \cdot \frac{(n + d)^{d-1}}{(d-2)! \cdot d} \approx \frac{d!}{(n + d)^d} > \frac{1}{(n \choose d)}. \tag{7.10}
\]

Hence, the number of rounds is significantly larger using Subspace Reduce. For the makespan volume, the ratio is roughly

\[
\frac{\text{Makespan volume of Sparse Grid Reduce}(n, d)}{\text{Makespan volume of Subspace Reduce}(n, d)} \approx \frac{2 \cdot 8 \cdot \log_2 (n + d)}{2 \cdot 8 \cdot \log_2 (n + d)} \cdot \frac{2^{n-1} \cdot \frac{n^{d-1}}{(d-1)!}}{\frac{n^{d-1} \cdot 2^{n-1}}{(d-2)!} \cdot 2^{n-1}} = \frac{d}{d-1} \cdot \frac{n^{d-1}}{(n + d)^d} \approx \frac{n^{d-1}}{(n + d)^d} \quad n \to \infty. \tag{7.10}
\]

This means that the makespan volume for Sparse Grid Reduce and the original version of Subspace Reduce is roughly identical.
Keep in mind that these findings rely on rough estimates and that we are comparing two upper bounds. In particular, the bandwidth term for Subspace Reduce was overestimated using Equation 7.8. The estimate Equation 7.10 suggests that the makespan volume of Subspace Reduce is higher than that of Sparse Grid Reduce. As Subspace Reduce is designed to minimize the total communication volume this is a surprising result. In fact, the ratio of the makespan volumes of the precise formulas Equation 7.3 and Equation 7.4 for fixed level $n$ and dimension $d$ confirms that the makespan volume of Subspace Reduce is smaller. Furthermore, with increasing level and dimension this ratio grows slowly indicating that the makespan volume of Subspace Reduce decreases with respect to that of Sparse Grid Reduce. The ratio, however, seems to stay bounded by a constant depending only on the dimension $d$. Subspace Reduce handles increment spaces (almost) in serial (for the analysis we assumed strict serial reduction of the increment spaces) and hence it cannot take advantage of the reduced total communication volume.

This raises the question about the makespan volume of Parallel Subspace Reduce. We do not present, however, similar formulas for Parallel Subspace Reduce as the results for the original version of Subspace Reduce are already hard to read. In contrast, we refer the reader to the experimental part in Section 7.7, in which the number of rounds and the makespan volume are compared for Sparse Grid Reduce, Subspace Reduce and Parallel Subspace Reduce for specific dimensions $d$ and sparse grid levels $n$.

7.6 TESTBED AND EXPERIMENTAL SETUP

This section first describes the implementations of Sparse Grid Reduce and the original and parallelized versions of Subspace Reduce as well as the modifications done to Subspace Reduce such that non-blocking MPI routines can be used. Then it elaborates on the systems used for the experiments, how the communication model is applied to predict the runtime of the communication algorithms and states the latency and bandwidth values for each of the systems.

7.6.1 Implementation and Setup of Experiments

For our experiments we assume that each component grid is stored on one node of an HPC system. We further assume that all communication involved in the global combination step is performed by exactly one MPI process per node. This process has access to all hierarchical increment spaces $H_{\ell'}$ of the component grid $C_{\ell}$ of its node. The coefficients of each $H_{\ell'}$ are stored in a distinct array. Whenever experiments are conducted for sparse grids with boundary, the boundary is of level-1 type (see Section 7.4.7.1).
In certain experiments the minimum-level is increased proportional to the level. A practical reason is that this keeps the number of component grids in $C_n^d$ below a certain threshold as we only had access to about 500 compute nodes (with up to 32 processors each) for our experiments. In these cases, we state the number of component grids used, the dimension $d$ of the sparse grid and the minimum sparse grid level $n$ at which the experiments start as well as the minimum-level $\ell_{\text{min}}$ for this $n$. When the sparse grid level $n$ is increased by one, the sum over the minimum-level $\|\ell_{\text{min}}\|_1$ is also increased by one. Therefore, we increase $\ell_{\text{min}}$ component-wise in a round robin fashion. As an example, for $d = 5$ there are 456 component grids for $n = 7$ and $\ell_{\text{min}} = (1, 1, 1, 1, 1)$. When the minimum-level is increased, the minimum-level vector for $n = 8$ would be $(1, 1, 1, 1, 2)$, for $n = 9$ it would be $(1, 1, 1, 2, 2)$, for $n = 12$ it would be $(2, 2, 2, 2, 2)$, and so on. This way, the structure of the communication task stays the same for all levels. I.e., there is a bijective mapping between the nodes of the communication tasks for the different levels (and corresponding minimum levels) such that the same nodes need to communicate. Just the communication volumes change.

The measurements for the communication task of a sparse grid of a certain dimension $d$, level $n$ and minimum-level $\ell_{\text{min}}$ are repeated at least 3 times and until at least 0.1s have elapsed. The average communication times are reported. For the repetitions of the measurements of a fixed communication task the same set of communication nodes is always employed. When the communication task changes such that the number of component grids changes, i.e., when the minimum-level is kept constant, then a different set of communication nodes is used for each communication task. When the minimum-level is increased and hence the number of component grids and communication nodes remains constant, always the same set of nodes is used on SuperMUC. For Hermit, the same set of nodes was used for $d = 5$, $n = 7$ and $\ell_{\text{min}} = (1, 1, 1, 1, 1)$ and when the minimum-level was increased such that the number of communication nodes needed is always 456. When an increasing minimum-level was used on Hermit for $d = 5$, $n = 5$ and $\ell_{\text{min}} = (1, 1, 1, 1, 1)$ (126 nodes) and for $d = 10$, $n = 4$ and $\ell_{\text{min}} = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ (286 nodes), different sets of compute nodes were used for each communication task as provided by Hermit’s scheduling system.

7.6.1.1 Sparse Grid Reduce

Sparse Grid Reduce, as shown in Algorithm 7.1, is performed in three steps. First, the coefficients of all $H_k \subset SG_{n-1}^d$ are copied into a single buffer. For the $H_k$ which do not exist in the component grid $C_\ell$ of a particular process, i.e $H_k \notin C_\ell$, the buffer is filled with zeros. Then, the $\text{MPI\_Allreduce}$ function is executed on all MPI processes in order to perform the $\text{AllReduce}$ on the buffer. As a last step, the (combined) co-
coefficients are copied from the buffer to their corresponding $H_k$. For our experiments the runtime of Sparse Grid Reduce includes all three steps.

7.6.1.2 Subspace Reduce

The implementation of Subspace Reduce (Algorithm 7.2) requires an initialization phase. In the initialization phase an MPI Communicator is created for each increment space $H_k \in \mathcal{H}_{n-1}^d$. The MPI Communicator for the increment space $H_k$ contains all compute nodes that contain this increment space, i.e., all nodes $\ell$ for which $H_k \subseteq C_\ell$ holds. Thus, the size of the communicators ranges from $d + 1$ to $|C_n^d|$. We have observed that the creation of MPI Communicators is an expensive operation, requiring a considerable amount of time. To name an extreme example: creating the communicators for $d = 3$, $n = 16$ took 0.3 s on SuperMUC. For comparison, the actual communication took 0.03 s for Parallel Subspace Reduce and no boundary values. However, for practical applications the communicators only have to be created once. We can assume that the allocation of component grids to the nodes usually does not change for a large number of time steps. For time-dependent PDEs, for example, the communication step happens every few time steps. Typically, there are plenty of communication steps before a reassignment of the nodes might become necessary. Thus, the time for the initialization of the Subspace Reduce method can be neglected. For this reason, we did not include the initialization phase in the measurements. In certain circumstances it might, however, be necessary to deal with load balancing or resilience issues. This could require a reassignment of the component grids to the nodes.

When performing the actual Subspace Reduce, each process loops over all $H_k \in \mathcal{H}_{n-1}^d$. For the original Subspace Reduce, the increment spaces $H_k \in \mathcal{H}_{n-1}^d$ are arranged in lexicographic ordering. When there is a jump in the lexicographic order, it can be possible to reduce the increment space before and after the jump in parallel. Hence, the original Subspace Reduce version might reduce some increment spaces in parallel. For Parallel Subspace Reduce, the increment spaces have been reordered according to Section 7.4.6 (during the initialization phase) to allow a parallel execution. For both versions of Subspace Reduce: If the increment space $H_k$ is on the node of the process, the coefficients of $H_k$ are copied to the buffer of the process. Then, the MPI_Allreduce function is executed on all MPI processes included in the communicator of $H_k$ in order to perform the AllReduce-routine on the buffered coefficients. Afterwards, the (combined) coefficients are copied back from the buffer to $H_k$. When an increment space $H_k$ does not exist on a certain process, this process simply skips these operations for the missing increment space. The experiments measure the time to loop over all increment spaces $H_k \in \mathcal{H}_{n-1}^d$ and include the two copy operations.
As the MPI Implementation by Cray available on Hermit also offers non-blocking collective operations, we additionally use a modified version of Subspace Reduce employing the non-blocking MPI_Iallreduce-routine on Hermit. Using the non-blocking routines Subspace Reduce is implemented as follows. First, each process executes a loop over all $H_k \in \mathcal{H}_n^d$. If the increment space $H_k$ exists on that process, the increment space is copied into its private buffer and the non-blocking MPI_Iallreduce function is called. This function immediately returns. It is crucial that a distinct buffer is created for each $H_k$. After the loop, we execute the MPI_Waitall function in order to wait until all reduce operations have been executed. When all reduce operations have been finished, each process loops over all $H_k \in \mathcal{H}_n^d$ and extracts the (combined) coefficients of increment spaces $H_k$ that are contained in the component grid $C_\ell$ of the process.

7.6.2 Exact Modeling of the Communication Times

In Section 7.5.1, exact formulas for the makespan volume and the number of rounds of Sparse Grid Reduce (Equation 7.3) and Subspace Reduce (Equation 7.4 – assuming all increment spaces are reduced serially) were deduced for the case of sparse grids without boundary and without minimum-level. As these formulas were already rather involved, we did not generalize them for sparse grids with boundary, minimum-level or Parallel Subspace Reduce.

To get exact values for the number of rounds and the makespan volume for these cases, we take advantage of the implementation. Auxiliary files are created containing all necessary information to calculate the number of rounds and the makespan volume exactly. For Sparse Grid Reduce we only need to know the size of $SG_n^d$ and the number of component grids in $C_n^d$. For Subspace Reduce we need the level vector of each hierarchical increment space $H_\ell \in \mathcal{H}_n^d$, the size of the increment space and the number of component grids containing it. For Parallel Subspace Reduce we also output which increment spaces are on the same relative position within a hypercube and are hence reduced in parallel. As the number of component grids which contain an increment space is the same for all increment spaces of a group that is reduced in parallel, the runtime of a group is determined by reducing the largest increment space of the group. For the original version of Subspace Reduce we assume in the model that all increment spaces are handled strictly one after another (though there might be parallelism in the implementation when there is a “jump” in the level-vector.)

Please recall also the assumption that the AllReduce-routine is implemented by two sweeps through a binomial tree as discussed in Section 7.4. Note that some modern MPI implementations do not distinguish between reduce and broadcast steps and use more sophisticated
7.6 Testbed and Experimental Setup

<table>
<thead>
<tr>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
<td>$B_0$</td>
</tr>
<tr>
<td>Hermit</td>
<td>1.4 $\mu$s</td>
</tr>
<tr>
<td>SuperMUC</td>
<td>2.0 $\mu$s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_0$</td>
</tr>
<tr>
<td>Stampede</td>
</tr>
<tr>
<td>JUQUEEN</td>
</tr>
</tbody>
</table>

Table 7.2: Latency and bandwidth values used for the lower and upper bounds in the communication model for Hermit, SuperMUC, Stampede and JUQUEEN. For Hermit and SuperMUC, they have been measured; for Stampede and JUQUEEN reported data has been used.

This can then lead to an even smaller total number of rounds and makespan volume.

7.6.3 Systems used for Measurements

Experiments have been conducted on two systems, Hermit and SuperMUC. To obtain reliable data for our communication model we measured the latency and bandwidth for MPI messages of different sizes for both systems with a so-called ping-pong test. The test is performed on two nodes. Node 1 sends an MPI message with a certain size to node 2. As soon as node 2 has received the message, node 2 sends a message of same size back to node 1. This procedure is repeated multiple times to obtain reliable average results. The minimum latency $L_0$ can be deduced from the time it takes to send an MPI message of minimum size, for example a data payload of a single byte. The maximum bandwidth $B_0$ can be observed only for large message sizes, messages larger than a couple of hundred kilobytes. Both values are used to obtain the lower bound in the model and are listed in Table 7.2. Furthermore, these values were used for the estimated runtimes presented later in Table 7.4, Table 7.5, Table 7.6, Table 7.7 and Table 7.8.

To obtain upper bounds for the model, consider that the bandwidth of the system typically increases as the size of the messages grows. The effect that only a fraction of the full bandwidth is available for small messages is partially captured by the latency term. Regardless of the size of the message, the latency gives a minimum communication time. To reproduce the dependency of bandwidth on the message size, a larger latency $L_1$ is assumed. This allows one to set a (larger) minimum bandwidth $B_1$ threshold which is not attained for messages smaller than a
couple of kB. The upper bound pair $L_1$ and $B_1$ is chosen such that all times measured in the ping-pong test stay below the threshold given by the model and this pair of values. This second pair of latency and bandwidth values is then used to estimate an upper bound for the runtime of Sparse Grid Reduce and Subspace Reduce.

7.6.3.1 Hermit

Hermit is a Cray XE6 system located at the High Performance Computing Center Stuttgart (HLRS). The reported LINPACK performance is 831.4 TFlops/s. Hermit consists of 3552 nodes connected in a 3-dimensional torus network of Cray Gemini interconnects. Each node has two sockets equipped with AMD Interlagos CPUs. Each CPU has 16 cores and 16 MByte L3 Cache. There is 32 GByte DDR3 RAM available per node. For our experiments we used the Cray MPI in version 6.2.1 and GCC 4.8.2 for compilation. This version of Cray MPI offers non-blocking collective operations, like $\text{MPI\_Iallreduce}$, according to MPI Standard 3.0. We take advantage of these routines and conduct experiments with the non-blocking MPI routines on Hermit. Please refer to Section 7.6.1.2 for the details of the implementation.

On Hermit, latency and bandwidth of the MPI messages strongly depend on the position of the nodes in the 3-dimensional torus network. More hops in the network lead to higher latency and lower bandwidth. We measured a minimum latency between $1.4\ \mu$s and $8\ \mu$s and a maximum bandwidth between $1.7\ \text{GByte/s}$ and $6\ \text{GByte/s}$ depending on the position of the nodes in the network. This is in agreement with material presented by HLRS [And]. For Hermit the following values were chosen for the model: lower bound: $L_0 = 1.4\ \mu$s, $B_0 = 6\ \text{GByte/s}$; upper bound: $L_1 = 9\ \mu$s, $B_1 = 1\ \text{GByte/s}$.

7.6.3.2 SuperMUC

SuperMUC is located at the Leibniz-Rechenzentrum (LRZ), and with a LINPACK performance of 2897.0 TFlops/s it is currently the second fastest system in Germany. As of June 2014, SuperMUC is ranked number 12 on the top500 list. It consists of 9216 thin nodes suited for massively parallel applications and 205 fat nodes with a large shared memory for data intensive applications. The thin nodes are arranged in 18 island of 512 nodes each and the experiments were restricted to 1 island of thin nodes. The nodes within an island are connected by a fully non-blocking Infiniband 4xFDR10 network. Each thin node consists of two sockets with Intel Xeon E5-2680 (Sandy Bridge) CPUs and has 32 GB DDR3 RAM. Each CPU has 8 cores and 20 MB L3 Cache. For our experiments we used IBM MPI 1.3 and the Intel C++ Compiler 13.1.

As all nodes within an island on SuperMUC are connected by a fully non-blocking interconnect, each pair of nodes within an island can simul-
7.7 simulations and experiments

Simulators and experiments simultaneously communicate at the same bandwidth and latency. To decrease the variance of our measurements, we therefore only used the nodes within a single island.

Sending messages inside an island, we measured a minimum latency of 2 µs and a maximum bandwidth of 4.7 GByte/s. These measurements agree with data found by other researchers. For SuperMUC the following values were chosen for the model: lower bound: \( L_0 = 2 \ \mu s \), \( B_0 = 4.7 \ \text{G Byte/s} \); upper bound: \( L_1 = 4 \ \mu s \), \( B_1 = 1 \ \text{G Byte/s} \).

7.6.4 Additional Systems Employed to Predict Runtimes

We extend the predictions of our model to other Top500 systems with a different interconnect in Table 7.8, namely JUQUEEN and Stampede. We have not done any measurements on these systems, so we can only rely on data provided by training material of the compute centers.

7.6.4.1 Stampede

Stampede has a LINPACK performance of 5168.1 TFlops/s and is installed at the Texas Advanced Computing Center. The nodes are connected with 56 GBit/s Infiniband FDR in a 2-level fat-tree topology. The maximum bandwidth peaks below 6.4 GByte/s [Sch13]. The reported latency is between 1.1 µs and 2.54 µs depending on the number of switch hops. The values \( L_0 = 1.1 \ \mu s \) and \( B_0 = 6.4 \ \text{G Byte/s} \) were used for the model.

7.6.4.2 JUQUEEN

Juqueen is a BlueGene/Q System, and it is the fastest system in Germany with a LINPACK performance of 5008.9 TFlops/s. The nodes are connected in a five-dimensional torus network. For MPI messages the reported latency to the nearest neighbours is 1.7 µs and the maximum bandwidth is 1.8 GByte/s [Kum12].

7.7 simulations and experiments

This section presents the experimental results. First, we discuss how the number of rounds, the makespan volume and hence the predicted runtime of the different algorithms compare to each other. Then we present experiments on Hermit and SuperMUC which we back up with bounds obtained from the communication model and based on the measured latency and bandwidth for these two systems. We conclude with an outlook on the communication times of problems that are out of our experimental scope as they require too many nodes.

In summary, we are going to see that Sparse Grid Reduce performs best for small levels and dimensions where the overhead of communi-
cating the whole sparse grid is small and communicating in few rounds is advantageous. If either the dimension or the level grows, \textit{Subspace Reduce} outperforms \textit{Sparse Grid Reduce} as it communicates significantly less data. The reordering of the increment spaces enables \textit{Parallel Subspace Reduce} to further improve communication time when the blocking MPI-routines are used. Using the non-blocking routines on Hermit further speeds up \textit{Subspace Reduce} and \textit{Parallel Subspace Reduce}. In particular, \textit{Subspace Reduce} and \textit{Parallel Subspace Reduce} perform very similar for the non-blocking routines. If the number of increment spaces and component grids grows (using a constant minimum level), the speedup of \textit{Parallel Subspace Reduce} increases as the level grows. If we increase the minimum level and hence fix the number of increment spaces and component grids, the speedup that \textit{Parallel Subspace Reduce} can achieve over \textit{Subspace Reduce} will be limited.

7.7.1 Predicting Runtimes

This section analyzes the number of rounds and the makespan volume for \textit{Sparse Grid Reduce} and the original and parallel version of \textit{Subspace Reduce} for various settings. In the following tables, we present the volume in terms of the numbers of grid points. Thus, the value is independent from the data representation being used (in the context of scientific computing, real-valued or complex numbers with single or double precision would be common) which dictates the actual storage/message size in terms of bytes. For the runtime predictions and the experiments we assumed real-valued numbers and used double precision (8 bytes per grid point).

Table 7.3 shows the number of nodes that would be required to run the communication task for the given dimension \(d\) and level \(n\) if no minimum level is used, i.e. \(\ell_{\text{min}} = 1\). The large number of component grids for \(d = 5\) and \(d = 10\) and moderate to high levels is the reason why we limit ourselves to theoretical predictions in this subsection for higher dimensionalities and why we cannot reproduce these settings experimentally. The number of component grids is quickly exceeding the number of nodes we had available for the experiments on either HPC system. When running the actual experiments for \(d = 5\) and \(d = 10\) on Hermit and SuperMUC we need to employ an increasing minimum level to limit the number of required component grids and nodes. Still, we are going to rediscover the most important trends of this subsection in the actual experiments and for lower dimensions. As the experimental results with constant minimum level \((d = 3)\) as well as with increasing minimum level \((d = 5\) and \(d = 10)\) are going to fit very well with the bounds given by the model, we are confident that the additional predictions are good estimates even for the runtimes of the experiments that are beyond our resource limitations.
Table 7.3: Number of component grids and hence nodes that would be required to run the communication task for the given dimension $d$ and level $n$ if no minimum level is used, i.e. $\ell_{\text{min}} = 1$.

Table 7.4 shows the number of rounds and the makespan volume for Sparse Grid Reduce, Subspace Reduce and Parallel Subspace Reduce for sparse grids of different dimensions $d$ and refinement levels $n$ without boundary and without minimum level (i.e. $\ell_{\text{min}} = 1$). Furthermore, we depict the ratio between Sparse Grid Reduce and the original and parallel versions of Subspace Reduce with respect to the number of rounds and the makespan volume. For Sparse Grid Reduce, the number of rounds is low for low levels and only increases slowly with increasing level $n$. In contrast, both versions of Subspace Reduce need significantly more rounds, and this number explodes with increasing level. Comparing the two extreme examples $d = 3$, $n = 5$ and $d = 10$, $n = 12$, the number of rounds increases only moderately for Sparse Grid Reduce from 10 to 38 while it explodes from 128 to $2.3 \cdot 10^6$ for the original Subspace Reduce. For $d = 5$ and $n = 20$, Parallel Subspace Reduce needs a factor of about 6000 more rounds than Sparse Grid Reduce. The reason for this extreme increase in the number of rounds for both versions of Subspace Reduce is the drastically increasing number of hierarchical increment spaces of the sparse grid, which are reduced individually. Regarding the makespan volume, the trends are reversed. The makespan volume of Sparse Grid Reduce is always larger than that of both Subspace Reduce versions. Furthermore, the makespan volume of Sparse Grid Reduce grows quicker than that of the Subspace Reduce versions. For $d = 5$ and $n = 20$, the makespan volume of Sparse Grid Reduce is 3.7 times higher than the volume of the original Subspace Reduce method. It is even 71 times higher than the volume of Parallel Subspace Reduce.
<table>
<thead>
<tr>
<th>$d = 3$</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
</tr>
<tr>
<td>$n = 5$</td>
<td>10</td>
<td>1110</td>
<td>128</td>
<td>582</td>
<td>104</td>
<td>390</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>16</td>
<td>303088</td>
<td>1400</td>
<td>111860</td>
<td>880</td>
<td>33908</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>18</td>
<td>2.71e+07</td>
<td>5588</td>
<td>9.28e+06</td>
<td>3186</td>
<td>1.45e+06</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>20</td>
<td>1.8e+09</td>
<td>14836</td>
<td>5.67e+08</td>
<td>8016</td>
<td>5.14e+07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
</tr>
<tr>
<td>$n = 5$</td>
<td>14</td>
<td>4914</td>
<td>434</td>
<td>2414</td>
<td>314</td>
<td>1454</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>22</td>
<td>4.13e+06</td>
<td>12736</td>
<td>1.42e+06</td>
<td>6586</td>
<td>321234</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>28</td>
<td>8.77e+08</td>
<td>100284</td>
<td>2.49e+08</td>
<td>45818</td>
<td>2.57e+07</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>30</td>
<td>9.68e+10</td>
<td>446574</td>
<td>2.63e+10</td>
<td>190776</td>
<td>1.37e+09</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 10$</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
</tr>
<tr>
<td>$n = 4$</td>
<td>18</td>
<td>4338</td>
<td>598</td>
<td>2058</td>
<td>526</td>
<td>1770</td>
</tr>
<tr>
<td>$n = 8$</td>
<td>30</td>
<td>1.19e+07</td>
<td>86948</td>
<td>3.71e+06</td>
<td>43544</td>
<td>1.16e+06</td>
</tr>
<tr>
<td>$n = 12$</td>
<td>38</td>
<td>4.85e+09</td>
<td>2.29e+06</td>
<td>1.26e+09</td>
<td>907862</td>
<td>2.06e+08</td>
</tr>
</tbody>
</table>

Table 7.4: Without boundary and without minimum level (i.e. $\ell_{min} = 1$): rounds and makespan volume (MkVol) for different dimensions. The makespan volume is expressed in the number of grid points (not bytes).
Comparing the two versions of *Subspace Reduce* one notes that the parallel version can only reduce the number of rounds by about a factor of 2 for large levels compared to its original counterpart. However, *Parallel Subspace Reduce* decreases the makespan volume significantly by a factor of around 11 for \( d = 3, n = 20 \), by a factor of around 19 for \( d = 5, n = 20 \) and a factor of about 6 for \( d = 10, n = 12 \) compared to the original *Subspace Reduce*. The enormous number of hierarchical increment spaces for large level or dimension enables a high degree of parallelism and thus a significant reduction in the makespan volume for *Parallel Subspace Reduce*.

Table 7.5 concerns sparse grids with boundary points and compares the number of rounds and the makespan volume for *Sparse Grid Reduce*, *Subspace Reduce* and *Parallel Subspace Reduce* for \( d = 5 \) and different \( n \). Including boundary points has the effect that the volume of the sparse grid and of the increment spaces which contain the boundary points is drastically increased. The number of increment spaces and the whole structure of the communication task is, however, the same as for sparse grids without boundaries of the same dimension and level. Thus for all methods, the number of rounds is identical to the case where no boundary points are used.

Due to the increased volume of the increment spaces containing the boundary points, the makespan volume is larger for all methods, though. The increase in makespan volume is as high as several orders of magnitude for small level. The makespan volume of *Sparse Grid Reduce* and *Subspace Reduce* increases almost by the same factor as it can be seen when comparing the ratio of their makespan volumes with and without boundary. In contrast, the makespan volume of *Parallel Subspace Reduce* increases faster than the makespan volume of the other two algorithms. For \( d = 5 \) and \( n \) ranging from 5 to 20, the ratio of the makespan volume between *Sparse Grid Reduce* and *Parallel Subspace Reduce* grows from 2.6 to 48, whereas it increases from 3.4 to 71 for sparse grids without boundary (see Table 7.4 for the latter one). This is due to the fact that increment spaces of small level sum contain a lot of boundary points compared to interior points and also contribute to many component grids. The increment spaces whose volume increases by the largest factors are those that cannot be reduced in parallel or only to a limited extend. Only the increment spaces with a high level sum can be reduced in parallel. But increment spaces of high level sum contain only relatively few boundary points in comparison to interior points.
Table 7.5: With boundary and without minimum level (i.e. $\ell_{\text{min}} = 1$): rounds and makespan volume (MkVol) for $d = 5$. The makespan volume is expressed in the number of grid points (not bytes).

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
</tr>
<tr>
<td>$n = 5$</td>
<td>14</td>
<td>168462</td>
<td>434</td>
<td>89802</td>
<td>314</td>
<td>63882</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>22</td>
<td>4.33e+07</td>
<td>12736</td>
<td>1.57e+07</td>
<td>6586</td>
<td>4.61e+06</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>28</td>
<td>5.23e+09</td>
<td>100284</td>
<td>1.53e+09</td>
<td>45818</td>
<td>2.22e+08</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>30</td>
<td>4.11e+11</td>
<td>446574</td>
<td>1.14e+11</td>
<td>190776</td>
<td>8.61e+09</td>
</tr>
</tbody>
</table>

Table 7.6: With boundary and with increasing minimum level: rounds and makespan volume (MkVol) for $d = 5$. Due to the increasing minimum level 456 component grids are always used. The makespan volume is expressed in the number of grid points (not bytes).

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Subspace Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>456 nodes</td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
<td>MkVol</td>
<td>Rounds</td>
</tr>
<tr>
<td>$n = 7$</td>
<td>18</td>
<td>1.85e+06</td>
<td>2184</td>
<td>795006</td>
<td>1324</td>
<td>403326</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>18</td>
<td>1.08e+07</td>
<td>2184</td>
<td>4.59e+06</td>
<td>1324</td>
<td>2.25e+06</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>18</td>
<td>2.32e+08</td>
<td>2184</td>
<td>9.8e+07</td>
<td>1324</td>
<td>4.61e+07</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>18</td>
<td>5.97e+09</td>
<td>2184</td>
<td>2.51e+09</td>
<td>1324</td>
<td>1.15e+09</td>
</tr>
</tbody>
</table>
Table 7.6 covers the case of increasing minimum-level and depicts the number of rounds and the makespan volume for Sparse Grid Reduce and Subspace Reduce and Parallel Subspace Reduce for $d = 5$ with boundary points. The setting of increasing minimum-level and including boundary points is also the crucial setting in which all experiments besides for $d = 3$ have been carried out in. The minimum-level is increased as described in Section 7.6.1 so that a constant number of 456 component grids are used for each level $n$. We started with $\ell_{\text{min}} = (1,1,1,1,1)$ for $n = 7$. When the minimum level is increased, the structure of the communication tasks stays the same for all levels $n$, only the communication volumes change. Therefore, the number of rounds is constant for different $n$ for all three methods and identical to the case of $n = 7$. As before, the number of rounds for Sparse Grid Reduce is significantly smaller than for Subspace Reduce and Parallel Subspace Reduce respectively, and Parallel Subspace Reduce decreases the number of rounds by roughly a factor of 2 compared to the original Subspace Reduce method.

The makespan volume is reduced by Subspace Reduce by roughly a factor of 2 and by Parallel Subspace Reduce by roughly a factor of 5 compared to Sparse Grid Reduce. However, the makespan volume ratio between Sparse Grid Reduce and Subspace Reduce, and Sparse Grid Reduce and Parallel Subspace Reduce respectively, increases only slightly with increasing $n$. This differs from the observations for fixed $\ell_{\text{min}}$ (see Table 7.5) where this ratio increased significantly as $n$ grew, in particular for Parallel Subspace Reduce. When using a minimum level, the communication task is much smaller than in case without minimum level. In particular, the number of hierarchical increment spaces remains constant and the structure of the communication task remains unchanged if the minimum level is increased simultaneously with the level $n$. This means that although the level vectors $\ell$ of the increment spaces participating in the communication task are increasing when $n$ grows, the differences $\ell - \ell_{\text{min}}$ remain unchanged. This relative level $\ell - \ell_{\text{min}}$, however, determines which increment spaces can be reduced in parallel. Thus, the amount of parallelism does not grow with $n$, as it was the case when no minimum level was used. If the minimum level is increased, Parallel Subspace Reduce does not significantly reduce the makespan volume compared to its original counterpart even for large $n$. All in all, due to Subspace Reduce and Parallel Subspace Reduce reducing the makespan volume by a small, almost constant factor compared to Sparse Grid Reduce, we also expect small, constant speedups for Subspace Reduce and Parallel Subspace Reduce over Sparse Grid Reduce when the minimum level is increased in the experiments.

For the rest of this section we limit the discussion to Sparse Grid Reduce and Parallel Subspace Reduce and disregard the original Subspace Reduce.
Table 7.7 shows the expected runtime of \textit{Sparse Grid Reduce} and \textit{Parallel Subspace Reduce} on Hermit, using the lower bound values $L_0$ and $B_0$, as well as the ratio of time spent in the latency and bandwidth term. This experiment was conducted including boundary points and using no minimum-level. For both methods the percentage of time spent in the bandwidth term grows with increasing level and dimension. However, for \textit{Sparse Grid Reduce} it starts off at a larger percentage and grows faster: almost all time is spent in the bandwidth term if either the dimension or the level is at least moderate. \textit{Subspace Reduce} spends a significant amount of time in the latency term due to the large number of rounds. However, the bandwidth term still dominates for large levels, or medium level and high dimension. While \textit{Sparse Grid Reduce} is observed to be faster for small to medium dimension and small level, \textit{Parallel Subspace Reduce} outperforms it in all other cases. In the first case, the volume overhead by communicating the complete sparse grid is not too large. It pays off that \textit{Sparse Grid Reduce} communicates in the minimal number of rounds. If either the dimension or the level grows, the size of the sparse grid grows significantly, such that \textit{Parallel Subspace Reduce} is predicted to outperform \textit{Sparse Grid Reduce} by one to two orders of magnitude.

Table 7.8 compares the predicted runtimes of \textit{Sparse Grid Reduce} and \textit{Parallel Subspace Reduce} on the four discussed systems Hermit, SuperMUC, Stampede and JUQUEEN for $d = 5$ using boundary points but no minimum level. Furthermore, two artificial systems are included which are derived from Hermit by either increasing or decreasing Hermit’s latency by a factor of 10. For the ratio of the expected runtimes of \textit{Sparse Grid Reduce} and \textit{Parallel Subspace Reduce} only the ratio between latency and bandwidth is important but not the actual latency and bandwidth values. Hence, increasing the latency by a factor of 10 has the same effect on this ratio as decreasing the bandwidth by a factor of 10.

For the four real systems the ratio of the runtimes between \textit{Sparse Grid Reduce} and \textit{Parallel Subspace Reduce} is very similar. On all four systems, the table shows that \textit{Parallel Subspace Reduce} is expected to outperform \textit{Sparse Grid Reduce} for $n \geq 10$. For $n = 20$, \textit{Parallel Subspace Reduce} is expected to be about 47 times faster than \textit{Sparse Grid Reduce}. Only when level and makespan volume are small ($n = 5$), \textit{Sparse Grid Reduce} is expected to perform best. If the latency of Hermit is reduced by a factor of 10, \textit{Parallel Subspace Reduce} becomes faster than \textit{Sparse Grid Reduce} even for $n = 5$. For small $n$, where a large fraction of the total time of \textit{Parallel Subspace Reduce} is spent in the latency term (see Table 7.7), \textit{Parallel Subspace Reduce} benefits from the reduced latency. In contrast, increasing the latency by a factor of 10 increases the runtime for \textit{Parallel Subspace Reduce} for $n = 5, 10, 15$. Note that for $n = 10, 15, 20$, changing the latency does not have a visible impact on the runtime of \textit{Sparse Grid Reduce}, as the fraction of time consumed by the latency term is negligible (see Table 7.7). According to Table 7.7 only the bandwidth term is relevant for
7.7 Simulations and Experiments

<table>
<thead>
<tr>
<th>$d = 3$</th>
<th>Sparse Grid Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Bandwidth</td>
<td>Total</td>
<td>Latency</td>
</tr>
<tr>
<td>$n = 5$</td>
<td>63.9 %</td>
<td>36.1 %</td>
<td>0.0000 s</td>
<td>97.4 %</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>2.1 %</td>
<td>97.9 %</td>
<td>0.0011 s</td>
<td>88.0 %</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>0.072 s</td>
<td>42.0 %</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>4.1 s</td>
<td>5.4 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th>Sparse Grid Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Bandwidth</td>
<td>Total</td>
<td>Latency</td>
</tr>
<tr>
<td>$n = 5$</td>
<td>8.0 %</td>
<td>92.0 %</td>
<td>0.0002 s</td>
<td>83.8 %</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>0.1 %</td>
<td>99.9 %</td>
<td>0.058 s</td>
<td>60.0 %</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>7.0 s</td>
<td>17.8 %</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>548 s</td>
<td>2.3 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 10$</th>
<th>Sparse Grid Reduce</th>
<th>Par. Subspace Reduce</th>
<th>Sparse Grid Reduce</th>
<th>Par. Subspace Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Bandwidth</td>
<td>Total</td>
<td>Latency</td>
</tr>
<tr>
<td>$n = 4$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>0.058 s</td>
<td>3.2 %</td>
</tr>
<tr>
<td>$n = 8$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>22 s</td>
<td>2.4 %</td>
</tr>
<tr>
<td>$n = 12$</td>
<td>0.0 %</td>
<td>100.0 %</td>
<td>2745 s</td>
<td>1.0 %</td>
</tr>
</tbody>
</table>

Table 7.7: With boundary and without minimum level (i.e. $\ell_{\min} = 1$): total runtime split into latency and bandwidth term for Hermit ($L_0 = 1.4 \mu s$, $B_0 = 6$ GByte/s) for different dimensions.

medium to high dimensions and levels. Whenever the dimension and the level are large enough – and hence the communication volume is large – then the latency is irrelevant and the runtime only depends on the bandwidth.

7.7.2 Results on Hermit

This section discusses the experiments conducted on Hermit. Unless we explicitly specify it, we always refer to the blocking implementations of Subspace Reduce and first compare those to Sparse Grid Reduce before also taking the non-blocking implementations into consideration.

Figure 7.3 shows the results on Hermit for $d = 3$ using no boundary points and a constant minimum level for $n = 3$ to $n = 18$, i.e., the number of nodes ranges from 10 to 460. For small $3 \leq n \leq 11$, Sparse Grid Reduce is faster than Subspace Reduce and Parallel Subspace Reduce. For small $n$ the increment spaces are very small (without boundary they are particularly small). Thus, the volume overhead by communicating the whole sparse grid is low. However, with growing $n$ the runtime of
### Table 7.8: With boundary and without minimum level (i.e. $\ell_{\text{min}} = 1$): runtime predictions for different systems and $d = 5$.

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th><strong>Hermit</strong></th>
<th><strong>SuperMUC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 5$</td>
<td>0.0002 s</td>
<td>0.0005 s</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>0.058 s</td>
<td>0.015 s</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>7.0 s</td>
<td>0.36 s</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>548 s</td>
<td>12 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th><strong>Stampede</strong></th>
<th><strong>JUQUEEN</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 5$</td>
<td>0.0002 s</td>
<td>0.0001 s</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>0.054 s</td>
<td>0.013 s</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>6.5 s</td>
<td>0.33 s</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>514 s</td>
<td>11 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d = 5$</th>
<th><strong>Hermit Latency decreased</strong></th>
<th><strong>Hermit Latency increased</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 5$</td>
<td>$L = L_0 / 10, \ B = B_0$</td>
<td>$L = 10 \cdot L_0, \ B = B_0$</td>
</tr>
<tr>
<td>$n = 5$</td>
<td>0.0002 s</td>
<td>0.0001 s</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>0.058 s</td>
<td>0.0071 s</td>
</tr>
<tr>
<td>$n = 15$</td>
<td>7.0 s</td>
<td>0.30 s</td>
</tr>
<tr>
<td>$n = 20$</td>
<td>548 s</td>
<td>12 s</td>
</tr>
</tbody>
</table>

Sparse Grid Reduce increases faster than for the other methods. It performs worse than Parallel Subspace Reduce for $n \geq 13$ and worse than Subspace Reduce for $n \geq 16$. For $n = 18$ the total time of Sparse Grid Reduce was 1.12s and the total time of Parallel Subspace Reduce was 0.10s, and hence Parallel Subspace Reduce is about 11 times faster than Sparse Grid Reduce. Parallel Subspace Reduce always performs better than its original counterpart. Furthermore, the difference between Subspace Reduce and Parallel Subspace Reduce grows with increasing $n$. The higher $n$, the more parallelism is possible due to the larger number of increment spaces. Thus, the difference in makespan volume between Subspace Reduce and Parallel Subspace Reduce grows with increasing $n$. This agrees with the predictions from Table 7.4. The two non-blocking variants of Subspace Reduce only show a slight improvement over blocking Parallel Subspace Reduce. Furthermore, there is no visible difference between the
two non-blocking variants. This means that reordering the communication tasks had no impact on the performance for the two non-blocking implementations of Subspace Reduce. The upper and lower bounds of the model predict the measurements very well. Except for the two outliers of Sparse Grid Reduce at \( n = 13 \) and \( n = 14 \), all results of the blocking implementations are within the bounds.

Figure 7.4 shows the results with boundary points for \( d = 3 \) and \( n = 3 \) to \( n = 16 \) for which the number of nodes ranges from 10 to 361. As in the case without boundary, Sparse Grid Reduce is faster than the other methods for small \( n \). However, in the region \( 3 \leq n \leq 9 \) the difference between Sparse Grid Reduce and Parallel Subspace Reduce is considerably smaller than in the case without boundary points. Furthermore, the crossover point between Sparse Grid Reduce and Parallel Subspace Reduce shifted from \( n = 12 \), for the case without boundaries, to \( n = 10 \). Due to the increased size of the increment spaces containing the boundary points, also the volume overhead when communicating the whole sparse grid increases. With growing \( n \) the runtime of Sparse Grid Reduce increases faster than for the other methods. Subspace Reduce performs better than Sparse Grid Reduce for \( n \geq 15 \) and Parallel Subspace Reduce performs better than Sparse Grid Reduce for \( n \geq 11 \). For \( n = 16 \) the total time of Sparse Grid Reduce was 0.46 s and the total time of Parallel Subspace Reduce was 0.059 s. Hence, Parallel Subspace Reduce is about 7.8 times faster than Sparse Grid Reduce. Comparing Subspace Reduce and Parallel Subspace Reduce we can observe the same as before: the parallel version is always faster and the difference in runtime grows with \( n \) due to the increasing degree of parallelism. As observed in the case without boundaries, the non-blocking variants show only a slight improvement over Parallel Sub-
space Reduce and, again, there is no noticeable difference between the two non-blocking variants. Except for the two outliers of Subspace Reduce at \( n = 12 \) and \( n = 13 \), the model’s bounds fit very well with the results.

Figure 7.5 shows the results for \( d = 5 \) and boundary points on Hermit. The measurements were done for level \( n = 5 \) to \( n = 14 \) and the minimum level \( l_{\text{min}} \) was increased starting from \( l_{\text{min}} = 1 \) for \( n = 5 \) to obtain a constant number of 126 component grids and hence 126 nodes of Hermit. For small levels \( n = 5 \) and \( n = 6 \), Sparse Grid Reduce was slightly faster than Parallel Subspace Reduce but as \( n \) grows, the runtime of Sparse Grid Reduce increases faster than the runtime of the other methods. This results in similar runtime of Sparse Grid Reduce and Parallel Subspace Reduce for \( 7 \leq n \leq 9 \) and Parallel Subspace Reduce performing better than Sparse Grid Reduce for \( n \geq 10 \). For \( n = 14 \), the runtime of Sparse Grid Reduce was 0.21 s and while Parallel Subspace Reduce executed in 0.026 s, a factor of about 8.1 with respect to runtime. Subspace Reduce performs better than Sparse Grid Reduce for \( n \geq 11 \). Parallel Subspace Reduce performed always better than Subspace Reduce, except for \( n = 6 \) where the two show no visible difference. The non-blocking variants of Subspace Reduce show a significant improvement over Parallel Subspace Reduce. For \( n = 14 \) the total time of non-blocking Subspace Reduce was 0.0073 s, which is 29 times faster than Sparse Grid Reduce. Unlike it was observed for \( d = 3 \), here even for small \( n \) the non-blocking variants of Subspace Reduce clearly outperform Sparse Grid Reduce. In this experiment there was a slight difference in runtime for the two non-blocking variants. Reordering the increment spaces had a small negative effect on the performance for most \( n \). Furthermore, the model fits very well with
the experimental results as all measurements are within the upper and lower bounds of the model.

Figure 7.6 shows the results for \( d = 5 \) with boundary points on Hermit. The measurements were done for level \( n = 7 \) to \( n = 13 \) and the minimum level \( \ell_{\text{min}} \) was increased starting from \( \ell_{\text{min}} = 1 \) for \( n = 7 \) to obtain a constant number of 456 component grids. Sparse Grid Reduce was slightly faster than Parallel Subspace Reduce only for the smallest level \( n = 7 \). As the runtime of Sparse Grid Reduce increases fastest with growing level \( n \), the crossover point between Sparse Grid Reduce and Subspace Reduce shifted from \( n = 11 \), where it was for 126 nodes, to \( n = 9 \) such that Sparse Grid Reduce performs worst for \( n \geq 9 \). Having started at a higher level \( n \), the number of increment spaces and also the total size of the sparse grid increased in comparison to the previous case with 126 nodes. Thus, also the overhead for Sparse Grid Reduce when communicating the whole sparse grid is higher than in the case with 126 nodes. For \( n = 13 \) the total time of Sparse Grid Reduce was 0.34 s and the total time of Parallel Subspace Reduce was 0.04 s, which is a factor of 8.5. Comparing Subspace Reduce and Parallel Subspace Reduce, the difference in runtime is considerably larger than with 126 nodes, since the higher number of increment spaces enables more parallelism. Unlike it was observed for \( d = 3 \) and a growing number of component grids, the difference between Subspace Reduce and Parallel Subspace Reduce does not increase with \( n \). Due to the increasing \( \ell_{\text{min}} \), the communication task remains the same independent of \( n \). Note that this behavior fits well to the predictions in Table 7.6, where Parallel Subspace Reduce has roughly half the number of rounds and half the makespan volume as Subspace Reduce independent of \( n \). As with 126 nodes, the non-blocking vari-
Figure 7.6: Results on Hermit for $d = 5$ and boundary points. $\ell_{\text{min}}$ increasing from $(1,1,1,1,1)$ for $n = 7$ to $(3,2,2,2,2)$ for $n = 13$, i.e. a constant number of 426 component grids was used.

ants of Subspace Reduce are considerably faster than the other methods. For $n = 13$ the total time of non-blocking Subspace Reduce was 0.0047 s, which is 72 times faster than Sparse Grid Reduce. Again, we can see that the parallel ordering of the increment spaces had a small adverse effect on the performance. Furthermore, the model’s lower and upper bounds fit well to the experimental results.

Figure 7.7 shows the results for $d = 10$ with boundary points on Hermit. The measurements were done for level $n = 4$ to $n = 8$ and the minimum level $\ell_{\text{min}}$ was increased starting from $\ell_{\text{min}} = 1$ for $n = 4$ to obtain a constant number of 286 component grids. Unlike observed for $d = 3$ and $d = 5$, Subspace Reduce and Parallel Subspace Reduce perform significantly better than Sparse Grid Reduce for all $n$, without exception. For $d = 10$ and boundary points the individual increment spaces are very large which results in a high volume overhead when communicating the whole sparse grid. For $n = 8$ the total time of Sparse Grid Reduce was 1.45 s and the total time of Parallel Subspace Reduce was 0.22 s, which is a factor of 6.6. As already observed for $d = 5$, the speedup between Parallel Subspace Reduce and Subspace Reduce remains nearly constant with increasing $n$. As we increase the minimum level, the number of increment spaces does not increase with growing level $n$ and hence increasing $n$ does not enable more parallelism. The non-blocking variants of Subspace Reduce were considerably faster than Subspace Reduce, although the difference seems to decrease with growing $n$. For $n = 8$ the total time of the non-blocking Parallel Subspace Reduce was 0.091 s, which is 16 times faster than Sparse Grid Reduce. Unlike observed for $d = 5$, there is no visible difference in runtime between the two non-blocking variants. All the experimental results are within the upper and lower bounds of
the model. However, *Parallel Subspace Reduce* performs very close to the lower bound.

### 7.7.3 Results on SuperMUC

In the following, the experimental results on SuperMUC are presented. No experiments were performed with the non-blocking variants of *Subspace Reduce*, as there was not yet an MPI implementation available which offered the non-blocking collective operations of the MPI 3.0 standard when we conducted this research.

Figure 7.8 shows the results for $d = 3$ without boundary and with constant minimum level $\ell_{\text{min}} = 1$ on SuperMUC for $n = 3$ to $n = 17$. The number of nodes ranges from 10 to 409. For small $3 \leq n < 13$, *Sparse Grid Reduce* performs best. With the increment spaces being very small for small $n$ the volume overhead by communicating the whole sparse grid is low and *Sparse Grid Reduce* benefits from the minimal number of rounds. For $n > 13$, *Parallel Subspace Reduce* performs best. This behavior was already observed on Hermit (see Figure 7.3), except that the crossover point between *Sparse Grid Reduce* and *Parallel Subspace Reduce* occurred already at $n = 12$. Here, it occurs at $n = 13$. For $n = 17$ the total time of *Sparse Grid Reduce* was 0.38 s and the total time of *Parallel Subspace Reduce* was 0.064 s, which is a factor of 5.9. As already observed on Hermit, the difference between *Subspace Reduce* and *Parallel Subspace Reduce* grows with increasing $n$ as a higher level $n$ enables more parallelism for *Parallel Subspace Reduce*. Unlike observed on Hermit, there is no clear trend of *Sparse Grid Reduce* becoming slower than *Subspace Reduce* with increasing $n$. The lower bounds of the model for *Subspace Reduce*
Figure 7.8: Results on SuperMUC for $d = 3$ without boundary and without minimum level ($\ell_{\text{min}} = 1$). Thus, the number of nodes increases with $n$.

Reduce and Parallel Subspace Reduce are slightly violated for $3 \leq n \leq 8$. The lower bound of Sparse Grid Reduce is violated for $10 \leq n \leq 13$. Nevertheless, the model covers the trends of the measurements very well.

Figure 7.9 shows the results for $d = 5$ with boundary points on SuperMUC. The measurements were done for level $n = 5$ to $n = 13$ and the minimum level $\ell_{\text{min}}$ was increased starting from $\ell_{\text{min}} = 1$ for $n = 5$ to obtain a constant number of 456 component grids. The results for Sparse Grid Reduce are very similar to those on Hermit. For small level ($n = 7$ and $n = 8$), Sparse Grid Reduce performs better than Parallel Subspace Reduce. For $n \geq 9$, Sparse Grid Reduce performs worse than Subspace Reduce. For $n = 13$ the total time of Sparse Grid Reduce was 0.28 s and the total time of Parallel Subspace Reduce was 0.037 s, which is a factor of 7.6. The ratio of the runtimes between Sparse Grid Reduce and Subspace Reduce is nearly constant for $n \geq 9$. This is in agreement with the predictions in Table 7.6, where the ratios between Sparse Grid Reduce and Subspace Reduce of both the number of rounds and the makespan volume are almost constant as well. Except for the two outliers of Sparse Grid Reduce at $n = 7$ and $n = 8$, all experimental measurements are within the model’s bounds.

Figure 7.10 shows the results for $d = 10$ with boundary points on SuperMUC. The measurements were done for level $n = 4$ to $n = 8$, and the minimum level $\ell_{\text{min}}$ was increased starting from $\ell_{\text{min}} = 1$ for $n = 4$ to obtain a constant number of 286 component grids. As already observed for $d = 10$ on Hermit, Sparse Grid Reduce performed worse than the other methods for all $n$. For $d = 10$, the individual increment spaces are already very large which results in a high volume overhead when communicating the whole sparse grid. For $n = 8$, the total time of Sparse Grid Reduce was 1.03 s and the total time of Parallel Subspace Reduce was 0.15 s,
which is a factor of 6.9. The runtime of *Subspace Reduce* is predicted correctly by the model as is the runtime of *Sparse Grid Reduce* besides one outlier for \( n = 5 \). The runtimes of *Parallel Subspace Reduce*, however, are considerably below the bound of the model. For \( d = 10 \), we already observed on Hermit that *Parallel Subspace Reduce* runs very close to the lower bound of the model. One explanation could be that more parallelism than taken into account by the model is possible. Another explanation could be that the *MPI_Allreduce* function is not implemented as a binomial tree as assumed by our model, but in a more efficient way.

### 7.7.4 Extending the Scope to Higher Levels

In this section we finally extend our predictions for SuperMUC into regions which are beyond what can be measured on current hardware. This is especially interesting to make predictions of the algorithms’ performance on future hardware. We consider a 5- and a 10-dimensional problem with boundary points and constant minimum level \( \ell_{\text{min}} = 1 \). Note that for \( d = 5 \) and \( n = 25 \) we would need 76,251 nodes and for \( d = 10 \) and \( n = 13 \) we would even need 646,580 nodes. For comparison, currently none of the world’s largest supercomputers has more than 100,000 nodes.

In Figure 7.11 we can see the predicted runtimes for \( d = 5 \) with boundary and without minimum level (i.e. \( \ell_{\text{min}} = 1 \)) for \( 7 \leq n \leq 25 \). For \( n = 7 \), all algorithms are expected to need roughly the same amount of time. With increasing \( n \) the difference between *Sparse Grid Reduce* and *Parallel Subspace Reduce* grows significantly. For \( n = 25 \) the predicted runtime of *Sparse Grid Reduce* is around 100,000 s (almost 28 hours) while
Figure 7.10: Results on SuperMUC for $d = 10$ with boundary, $\ell_{\text{min}}$ increasing from $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ for $n = 4$ to $(2, 2, 2, 2, 1, 1, 1, 1, 1, 1)$ for $n = 8$, resulting in a constant number of 286 component grids.

the predicted runtime of *Parallel Subspace Reduce* is only around 1000 s (less than 17 minutes). The execution time of *Sparse Grid Reduce* is so high that it would just not be reasonable to use this algorithm for actual computations. The difference between *Sparse Grid Reduce* and *Subspace Reduce* also increases with increasing $n$, but only slightly. For $n = 25$, *Subspace Reduce* would need more than 10,000 s and is thus less than a factor of 10 faster than *Sparse Grid Reduce*.

Figure 7.12 shows the predicted runtimes for $d = 10$ with boundary and without minimum level for $4 \leq n \leq 13$. For $d = 10$, it already required a considerable amount of time just to compute the predictions due to the extreme number of hierarchical increment spaces. Thus, we limited the use of the model to $n = 13$. The general trend is very similar to the case of $d = 5$. While all algorithms need almost the same time for $n = 4$, *Parallel Subspace Reduce* becomes the fastest and *Sparse Grid Reduce* the slowest algorithm as the level increases. For $n = 13$, *Sparse Grid Reduce* would require more than 10,000 s. In the same situation, *Parallel Subspace Reduce* would only require between 400 and 1100 seconds, and it would be 10 times faster. It is expected that this gap grows further as the level increases. As in the case for $d = 5$, the difference between *Sparse Grid Reduce* and *Subspace Reduce* grows much slower with increasing $n$.

7.8 Conclusions and Future Work

The combination technique provides an hierarchical approach to avoid the need for global synchronization for the numerical treatment of high-dimensional problems, for example PDEs such as they appear in plasma physics. It retrieves a solution as a suitable combination of several an-
Figure 7.11: Predictions for SuperMUC for $d = 5$ with boundary and without minimum level (i.e. $\ell_{\text{min}} = 1$). Thus, the number of component grids increases with $n$.

Figure 7.12: Predictions for SuperMUC for $d = 10$ with boundary and without minimum level (i.e. $\ell_{\text{min}} = 1$). Thus, the number of component grids increases with $n$. 
isotropic, coarse grid solutions which can be computed independently. After every few time steps of time-dependent problems, the partial solutions have to be combined in a reduce/broadcast step. In this chapter, we have studied this remaining synchronization bottleneck.

Two different algorithms, *Sparse Grid Reduce* and *Subspace Reduce*, have been described, and parallel and non-blocking variants have been discussed. Furthermore, we have given a precise model and cost measures that reasonably reflect modern architectures, allowing us to analyze the algorithms and prove lower bounds for the communication step. The model has been well confirmed using numerical experiments on several HPC systems. The model and the experiments fit very well, especially on Hermit, despite the simplifying assumption that the *AllReduce*-operations work in two phases and that these phases are based on binomial trees for communication. Even more, the model can provide predictions for HPC systems that do not exist as yet and for problem sizes that are currently still out of scope.

For large levels, when the sparse grid is significantly larger than any component grid, *Subspace Reduce* outperforms the naive approach *Sparse Grid Reduce*. This effect can already be observed if *Subspace Reduce* reduces the hierarchical increment spaces in a serial manner. The parallel and non-blocking variants of *Subspace Reduce* improve that even further, up to orders of magnitude depending on the problem size. Nevertheless, there still is room for improvement in future work. *Subspace Reduce* can likely be improved by merging small hierarchical increment spaces and communicating them jointly. Merging small increment spaces would decrease the number of rounds significantly while only slightly increasing the makespan volume.

For the analysis, it was assumed that there is precisely one communication node per component grid. Future work can consider solving one component grid in parallel on many communication nodes, using one communication node to solve several component grids or a combination of both. Furthermore, the possibility of using additional communication nodes that do not solve partial solutions can be explored.

For the experiments, a selection of the component grids based on a minimum level was used. This limits the number of component grids to the available compute resources. For larger numbers of compute nodes, much higher speedups of parallel *Subspace Reduce* are predicted by the communication model, and the use of the new communication algorithms will pay off even more.
Due to the large and still increasing gap between peak performance and memory bandwidth, communication-efficient algorithms have become more and more important. This effect is reinforced as the data sets that are processed continuously grow in size. In particular, high-dimensional applications create huge amounts of data that can only be processed if communication-efficient algorithms are at hand. In this thesis, communication refers to both data transfer through the memory hierarchy as well as communication via message passing.

The goal of this work was to optimize the communication behavior of existing numerical methods and to prove lower bounds for the respective problems. While a lot of effort is put into optimizing numerical algorithms, they are rarely studied from a theoretical point of view, in particular with respect to communication. Lower bounds for the communication requirements of the numerical methods are even rarer. This work has used theoretical models to analyze common algorithmic approaches and has shown that studying theoretical aspects may result in better algorithms from a practical point of view. First, the communication bottlenecks of the existing algorithmic approaches were identified. Then, the algorithms were optimized with respect to their communication behavior and, by doing so, new communication-efficient versions of these algorithms were created. In addition, lower bounds were proven that show that the new algorithms are either optimal or within a small constant factor of the optimum. All presented algorithms optimize the communication behavior of existing techniques, i.e., no new numerical methods were derived in this work. In addition, the sparse grid part of this work focuses on efficient algorithms for the component grids of the sparse grid combination technique and does not address algorithms for regular or adaptive sparse grids. Moreover, while the theoretical considerations immediately led to efficient implementations for two of the considered problems, further research should be conducted before the other two, theoretically optimal, algorithms are implemented.

This chapter continues with the description of the analyzed problems and gives a brief recap of the most important aspects of sparse grids and the combination technique. Then, the contributions of this thesis are discussed and used to draw conclusions. Subsequently, the limitations of this study are summarized and directions for future research are pointed out. Afterwards, this chapter finishes with concluding remarks.

This work has studied different numerical problems on full and sparse grids that arise when PDEs are solved. PDEs are one of the most im-
important classes of numerical problems. One particular application that motivates this thesis is the simulation of hot plasmas in fusion reactors for which a 5- or 6-dimensional PDE is solved for each time step. The problems studied in this thesis are, for low dimensions, stencil computations as they appear in finite difference methods. For high dimensions, this thesis focuses on the sparse grid combination technique.

Sparse grids are a hierarchical discretization scheme that enables a lessening of the “curse of dimensionality”, i.e., the exponential dependency of the number of grid points on the dimension of the problem. Sparse grids reduce the number of grid points by choosing a suitable basis, the hierarchical basis, and selecting the most important functions of this basis a priori. The algorithm performing this change of basis from the full grid basis to the hierarchical basis is called the hierarchization algorithm. Hierarchization is fundamental for sparse grids and prototypical for sparse grid algorithms, i.e., optimizations for the hierarchization algorithm are also likely to improve the performance of other sparse grid algorithms. Furthermore, hierarchization can be an important preprocessing step for the communication schemes that perform the communication step of the combination technique explained later. In this thesis, the hierarchization algorithm is analyzed and optimized for the component grids of the sparse grid combination technique.

The combination technique is an extrapolation scheme that splits a problem on a sparse grid into several, much smaller, independent subproblems on so called component grids. By doing so, the combination technique breaks the global communication requirements of conventional discretizations. In particular, the combination technique enables the computation of the component grid solutions in parallel and with standard solvers. A reduced communication is, however, still necessary. When the combination technique is applied to time-dependent PDEs, the sparse grid solution should be assembled from the component grid solutions every few time steps and this joint solution needs to be distributed back to the component grids. This remaining communication bottleneck of the combination technique was also studied in this work.

This thesis has restructured the computations of existing numerical methods to create memory-efficient algorithms. First, the problems were studied from a theoretical perspective. The insights from this analysis were used to identify the communication bottlenecks of existing algorithms and to rethink the present algorithmic solutions. For all problems that were discussed, novel algorithmic ideas were developed that avoid the respective bottlenecks and decrease communication.

- For stencil computations, band algorithms and data layouts that exactly match the access patterns of the band algorithms were derived to increase spatial as well as temporal locality.

- The implementation of the unidirectional hierarchization algorithm exploits the regular structure of the component grids to simplify
the navigation on the grid. Furthermore, it always works in the direction of the data layout, i.e., orthogonally to the poles for $d \geq 2$, to increase spatial locality.

- The divide and conquer hierarchization algorithm is cache-oblivious and avoids the unidirectional principle on a global scale. Instead, it applies it recursively to smaller subproblems. It thus reduces the $d$ global passes of the unidirectional principle to a single one.

- The presented communication scheme Subspace Reduce takes advantage of the hierarchical structure of sparse grids to minimize the total communication volume.

For all problems, lower bounds were proven that show that the derived algorithms are either optimal or within a small constant factor of the optimum. For the unidirectional hierarchization of component grids and the communication step of the combination technique, the theoretical considerations also guided the development of actual implementations.

It was hence demonstrated that the abstraction of the theoretical models provides a good framework to rethink algorithmic solutions and to come up with new algorithmic ideas. While the models make many simplifying assumptions, they are designed to capture the most important aspects with respect to communication. This enables one to focus on the most important aspects while not being distracted by minor details. Furthermore, the implementations show that theoretically optimal performance can be translated into high-performance code, and that the theoretical complexity of an algorithm can be used to predict the runtime of its implementation with high accuracy. In particular, the communication model presented in Chapter 7 only uses a latency and a bandwidth term to predict the time it takes to send a message. It was demonstrated that this model, despite its simplicity, predicts the runtime of the communication schemes accurately. This predictive power of the theoretical models justifies their usage, the simplifying assumptions they make and the focus on communication.

This work started with the assumption that communication is a crucial, or even the dominating factor influencing the runtime of many algorithms. To confirm this assumption, algorithms for two of the problems were implemented, and it was in fact discovered that their communication is crucial for their performance. Hence, while communication is not the only relevant issue for efficient algorithms, this thesis supports the point of view that communication is one of the crucial factors for performance. Furthermore, this work stresses the point that communication aspects should always be kept in mind when designing algorithms. Already prototype implementations can benefit from a communication-efficient implementation.
This thesis employs theoretical models that focus on communication to study the complexity of the considered problems. Hence, naturally, the theoretical analysis carried out in this thesis is limited to the communication requirements of the studied problems and the communication behavior of the derived algorithms. Furthermore, the models make several simplifying assumptions as discussed in Chapter 2. Therefore, these theoretical considerations can only be taken as a starting point. Of course, an efficient implementation also has to consider many more effects, which are not covered in the models that were used. For stencil computations as well as for the divide and conquer hierarchization algorithm, the algorithmic solutions were not implemented on purpose. While these algorithms minimize communication, they have a complicated structure which should be simplified before attempting an implementation. On the other hand, for the communication schemes and the unidirectional hierarchization algorithm, the theoretical analysis produced efficient approaches that were directly implemented.

In addition, all considered problems have a low operational intensity as they can either be phrased as sparse matrix-vector multiplication or concern a pure communication task. Thus, these problems are likely to be memory bound. There are also problems which are more compute-heavy and hence more likely to be compute-bound. If an algorithm solving a compute-heavy problem is implemented such that it achieves peak performance, reducing the communication of the respective algorithm cannot increase its performance any further. Still, there are many problems, including matrix-vector multiplication and sparse matrix-vector multiplication, whose operational intensity is bounded by a small constant. These problems are likely to benefit from algorithms that reduce communication.

Regarding sparse grid algorithms, this work has focused on the sparse grid combination technique and has optimized the hierarchization algorithms specifically for the component grids of the combination technique. Hierarchization algorithms for regular or adaptive sparse grids, were not considered explicitly. It is, however, possible to apply the divide and conquer approach for hierarchization to regular, adaptive and other kinds of sparse grids, as well.

Furthermore, this work has focused on the optimization of existing numerical methods. No new numerical schemes to solve the respective problems were derived. For that, a different focus would be required. While the numerical methods were the crucial starting points of this work, optimizations regarding communication can be used early in the design process of numerical methods. Once the convergence of a new numerical method has been established, its first implementation can already benefit from a communication-efficient design.

As the gap between peak performance and bandwidth is growing further, the design of communication algorithms will gain even more impor-
tance in the future. This work has studied a small selection of numerical problems with low operational intensity and developed communication-efficient solutions for those problems. While the chosen problems are general methods that can be applied to a variety of specific problems, like different PDEs, they cover a mere drop in the ocean of problems that can benefit from a communication-efficient implementation. In addition to considering other problems, there are also further research directions for the ones studied in this thesis. For the examined problems, future research directions cover the further development of the derived methods as well as the application of the derived insights to a more general class of algorithms.

Let us first discuss how the stencil algorithms as well as the divide and conquer hierarchization algorithm can be developed further such that their implementation becomes more realistic. For the stencil computations, the given insights can be used to derive optimizations for multiple steps of the stencil computation, i.e., multiple updates of the whole grid according to the stencil. As discussed in Section 3.9, the non-compulsory term would then dominate the complexity, and the derived optimizations would affect the leading term of the I/Os. For the divide and conquer hierarchization algorithm, instead of a cache-oblivious, a cache-aware approach shows promise, as discussed in Section 6.4. A cache-aware version would simplify the algorithm while presumably improving its performance further.

As the hierarchization algorithm is prototypical for sparse grid algorithms, the optimizations presented for hierarchization can also be applied to a more general class of sparse grid algorithms. The optimizations of Chapter 5 which were applied to the unidirectional hierarchization algorithm heavily exploit the regular structure of the component grids. Hence, those optimizations are mainly applicable to algorithms that are specialized in the component grids and relevant for the sparse grid combination technique. In contrast, the divide and conquer approach of Chapter 6 can be applied to any kind of sparse grid and sparse grid algorithm. Keep in mind that it is important for the divide and conquer approach that subgrids, even of regular and adaptive sparse grids, are stored compactly as discussed in Section 6.4. One class of sparse grid algorithms which might benefit from a divide and conquer approach are the so called UpDown schemes used to implement more complicated algorithms, such as PDE solvers, directly in the hierarchical sparse grid basis. As the UpDown scheme duplicates the grid and works on $2^d$ different copies of the grid in total, communication-efficient approaches are crucial due to the increased amount of data.

In summary, this work has demonstrated that I/O theory can guide the development of efficient numerical algorithms. In particular, this work has decreased the gap between I/O theory and existing numerical algorithms for a selection of problems. We have seen that it can be
worthwhile to take a step back and reconsider the problem to come up with a new algorithmic solution. While this approach may take longer than incrementally improving the algorithm, it also offers the opportunity of larger speed-ups. There are many other problems, that are also likely to benefit from this approach.

Furthermore, it was demonstrated that the cooperation between scientific computing, on one side, and theoretical computer science, on the other side, can be beneficial for both fields. Following such an approach, the scientific computing community can gain new perspectives to analyze their algorithms and an additional set of optimizations to speed up their implementations. The theoretical computer science community can acquire the chance to analyze and optimize problems that are important in practice. Moreover, the theory community can use these real-world problems to further develop and continuously improve and validate its theoretical models. For a successful cooperation between the two fields, the steady exchange of ideas is necessary. This needs to be encouraged if more interdisciplinary research is supposed to happen.

To conclude this work, let us consider the sparse grid combination technique as a whole: now, all important components are in place to solve high-dimensional PDEs efficiently with the combination technique, as it is depicted in Figure 4.2. The unidirectional hierarchization algorithm has been implemented almost optimally for component grids, and efficient communication schemes for the reduce/broadcast step have been derived and tested. Only an efficient implementation of the dehierarchization algorithm for the component grids is missing. Such an implementation is, however, straightforward given the implementation of the hierarchization algorithm. Hence, we are now ready to put the pieces together and solve high-dimensional PDEs with the combination technique, e.g., using the plasma physics code GENE to solve the gyrokinetic approach. This work can perhaps make a contribution to the development of fusion reactors and humanity’s quest for clean energy.
BIBLIOGRAPHY


COLOPHON

This document was typeset in \LaTeX using the typographical look-and-feel classicthesis. Most of the plots in this thesis were either generated with tikz, gnuplot or Mathematica. The bibliography was typeset using BibTex.