Efficient Regular Sparse Grid Hierarchization by a Dynamic Memory Layout

Riko Jacob

8.08.2013

Abstract

We consider a new hierarchization algorithm for sparse grids of high dimension and low level. The algorithm is inspired by the theory of memory efficient algorithms. It is based on a cache-friendly layout of a compact data storage, and the idea of rearranging the data for the different phases of the algorithm. The core steps of the algorithm can be phrased as multiplying the input vector with two sparse matrices. A generalized counting makes it possible to create (or apply) the matrices in constant time per row.

The algorithm is implemented as a proof of concept and first experiments show that it performs well in comparison with the previous implementation SG++, in particular for the case of high dimensions and low level.

1 Introduction

In many applications high dimensional models arise naturally, for example as a function that maps the high dimensional space $[0,1]^d$ to the real numbers. Such a function can be represented using a regular grid. If we use a regular grid that has N points in every direction, this approach leads to N^d sample points, which can be more than what is computationally feasible.

One approach to reduce the number of sample points are sparse grids introduced by Zenger [12] in 1991. To see the potential for improvement, consider an axis parallel line through a grid point. We call the sampled points that lie on such a line a *pole*. In the regular grid, all poles consist of N points. Take two poles a and b in direction e_d that are shifted in direction e_1 by $1/N$. For a smooth function f we would expect that the restricted functions $f|_a$ and $f|_b$ are very similar, at least for small $1/N$. Still, the full grid samples both functions on N points. In contrast, the sparse grid uses a different number of grid points on the two poles, say more on a than on b, and uses the samples for $f|_a$ to increase the accuracy of the representation of $f|_b$.

The sparse grid interpolation is based on hierarchical basis functions, as detailed for our concrete setting in Section 2. Each such basis functions has an axis parallel hyperrectangle as support and the volume of this support is $2^{-\ell}$, where the integer ℓ is the level of the basis function. Importantly, the support is not necessarily square, i.e., the side length can change drastically with the dimension. By using only basis functions of level up to $n \geq \ell$, we get some poles (actually one for each direction) that are sampled with $N = 2^{n+1} - 1$ points, whereas all other poles are sampled with fewer points. In contrast to the corresponding full grid with its N^d points, the sparse grid has only $O\left(N\binom{n+d-1}{d-1}\right) = O\left(N\left(\frac{e(n+d)}{d-1}\right)^{d-1}\right)$ sample points. The rate of convergence, i.e., how the approximation accuracy depends on increasing N, remains comparable to that of the full grid [12, 3, 4].

If we consider some examples, we see that the sparse grid with $N = 7$ points on the most densely sampled poles has for $d = 100$ only 20'401 grid points. For comparison, the corresponding full grid has $7^{100} \approx 10^{84}$ grid points, more than what is currently assumed to be the number of atoms in the universe. For $N = 7$ and $d = 10'000$ there are 200 million sparse grid points and for $N = 15$ and $d = 100$ there are 72 million. In this situation, a different asymptotic estimate is helpful, namely $O\left(N\binom{n+d-1}{d-1}\right) = O\left(N\binom{n+d-1}{n}\right) = O\left(N\left(\frac{e(n+d)}{n}\right)^{n}\right)$. Concretely, we see that for $N = 7$ the number of grid points grows with $\Theta(d^2)$, for $N = 15$ with $\Theta(d^3)$, and so on. Hence high dimensions might be numerically feasible for small N. In this work, we focus as an example on one particular task, namely the hierarchization of a sparse grid (see Section 2). For this particular task, there is one value per grid point as input and output, and this number of values is referred to as *degrees of freedom* (DoF). Further, hierarchization is a task of relatively small computational intensity, i.e., in every round of the algorithm every variable gives rise to at most 4 floating point operations. Hence our algorithmic ideas are related to good memory usage. On one hand this amounts to minimizing the size of the data structures (ideally only one variable per DoF), and on the other hand we want make sure that the data access patterns are cache-friendly. This leads us to so called *memory efficient* or *I/O-efficient* algorithms. While the above points are our main focus, clearly a useful implementation must reflect other aspects of modern hardware. One noteworthy aspect is parallelism, for example in the context of several cores of the same CPU.

Many efficient algorithms known for sparse grids are based on the *unidirectional principle* [3]. It allows us to operate only on the one-dimensional sparse grids, i.e., the poles. More precisely, we iterate over the dimensions and for each dimension consider each of the poles in this dimension on its own.

The question we consider in this work is how to implement the unidirectional principle I/O-efficiently, with the example of hierarchization. Let us pictorially describe the situation by thinking of the sparse grid as a work piece that needs to be drilled from many different directions. There are two standard ways of doing this: Either you mount the working piece to the bench and move a mobile drill around it, or you mount the drill on the bench and turn the working piece. We propose to switch to the analogue of the latter method: Instead of adapting the one-dimensional hierarchization procedure to the current dimension, we move the data. For the one-dimensional hierarchization algorithm to be I/O-efficient, it would be good if each pole we work on is stored contiguously in memory. Provided that each pole fits into the cache, we immediately get an I/O-efficient algorithm: It loads the pole once and efficiently because the pole is split into few cache lines. Then it performs all operations on this pole in the cache, and writes the finished pole back to main memory to free the cache.

Because it is impossible to have a data layout of the sparse grid that stores all poles of all dimensions contiguously, we rearrange the layout of the sparse grid according to the dimension in which we currently work. More precisely, we define a *rotation* of the sparse grid that is a cyclic shift of the dimensions, i.e., maps $(x_1, \ldots, x_d) \mapsto (x_2, x_3, \ldots, x_d, x_1)$. Using this, it is sufficient to be able to perform the one dimensional hierarchization algorithm efficiently in one of the dimensions. We chose for this *working step* to operate in dimension d.

This approach has four main advantages:

- We can choose a memory layout that makes the access pattern of the working step cache-friendly
- Both phases (working and rotation) are exactly the same for all d rounds. They can be phrased as sparse matrix multiplication. Computing these matrices once is sufficient.
- There is no need to store position information (like level and index) together with a variable. This role of the variable is always implied by the position of the variable in the array representing the sparse grid. This leads to a fairly small memory-footprint, in particular in comparison to hash-based implementations.
- The algorithm can be easily and efficiently (both computation and memory access-wise) parallelized for multiple cores.

1.1 Algorithmic Background

The theory of I/O-efficient algorithms go back to the definition of the I/O-model [1]. It is based on the idea that the CPU can only work on a memory of size M (the cache). Input, output and auxiliary storage are in *external memory*, which is unbounded in size and organized in *blocks* of size B (a cache-line). The running time of an algorithm is estimated by the number of *I/O-operations* that read a block from external memory or write a block to external memory.

The differences between the I/O-model and the somewhat similar RAM or von-Neumann model can be illustrated by considering the task of permuting. Given a permutation $\pi\colon \{1, \ldots, N\} \to \{1, \ldots, N\}$, a program for *permuting* takes an input vector $(x_1, \ldots, x_N)$ and creates the output vector $(y_1, \ldots, y_N)$ according to $y_{\pi(i)} = x_i$. The naive algorithm loops over the input and writes the value to the specified position of the output. On the RAM, this takes $\Theta(N)$ operations which is trivially optimal. The naive algorithm can be adapted to the I/O-model where it causes $\Theta(N)$ I/O-operations. In contrast to the RAM this is not always optimal. The alternative algorithm is to use a B/M-way merge sort to rearrange the input array into the output array, which takes $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. If the logarithmic term is smaller than B, this is better than the naive algorithm. When the I/O-model is used to describe a situation where the external memory is a disk and the internal memory is main-memory, the naive algorithm can easily be orders of magnitude slower. This is complemented by a lower bound stating that taking the better of the two mentioned algorithms is optimal [1]. The lower bound holds even if the algorithm can be adapted to the permutation, but this permutation is random. In contrast, there are also some permutations that can be performed easily, for example if the permutation is a cyclic shift or moves the elements not too far, e.g., $|\pi(i) - i| < M - 2B$. Many

RAM algorithms have a completely unstructured memory access, similarly to the naive algorithm for permuting. Sometimes this can be avoided by rearranging the data [7]. One example of this technique is the I/O-efficient algorithm for multiplying a dense vector with a sparse matrix [2]. The first phase of the algorithm is to create all elementary products by multiplying each entry a_{ij} of the matrix with x_j. To perform this I/O-efficiently the matrix should be stored in a column major layout such that matrix entries of the same column are stored together and the columns are in the same order as the input vector. In a second phase of the algorithm the output is computed as row sums of the elementary products. For this to be I/O-efficient, they are first rearranged into a row-major layout such that all entries belonging to the same row are stored together. Here, the I/O-efficient way to rearrange might be to use the sorting algorithm.

Note that even though we phrase our result here as a multiplication with a sparse matrix and a permutation matrix, the structure of these two particular matrices usually makes the naive algorithm I/O-efficient. Still, we use the idea of working in phases and rearranging the date between the phases.

1.2 Related Work

Many different aspects of sparse grids have been investigated in the last years. The presentation here follows [12, 3, 4, 10].

Most of the proposed algorithms for sparse grids have been implemented, and some of the code is publicly available.

One easily available implementation is SG++ [10, 11]. Its focus is adaptive sparse grids and it provides a lot of generality in terms of the used basis functions. Because of its availability and ease of installation, we use this for comparison in this paper.

The idea of using a layout of the complete sparse grid as a compact data structure has been proposed [8], but the layout itself differs from what we consider here. The corresponding implementation `fastsg` is described [9], where a recursive formula for the size of the sparse grid is proposed, similar to the one presented here. The code is publicly available but does not provide a hierarchization procedure for the 0-boundary case considered here.

Some of the ideas presented here (rotation, compact layout) are also the basis for a different implementation with focus on SIMD parallelism [5]. Optimizing for this kind of parallelism favors a different layout. This code has been extended with a focus on evaluation [6].

2 Sparse Grids

Sparse grids have been designed to approximate functions in high dimensional spaces with relatively few degrees of freedom. By now, there is a body of literature discussing the mathematics of sparse grids. In contrast, this work investigates only the computer science aspects arising in the context, actually only for one particular task called hierarchization. Nonetheless, there is the need for a brief discussion of the underlying mathematical concepts, at least to explain how we phrase them in our algorithms.

In the following $(0, 1) \subset \mathbb{R}$ denotes the open interval from 0 to 1, whereas $[0, 1]$ denotes the closed interval. A sparse grid space as we use it here is a finite dimensional linear space. Its dimension is phrased as *degrees of freedom* (DoF) and it is given by the size of the basis we define for the space. The elements of the basis are specific continuous functions $\mathbb{R}^d \to \mathbb{R}$ with support limited to $(0,1)^d$, the so called *hierarchical basis functions* as defined in the following. An element of the *sparse grid space* is a linear combination of these basis elements, and hence also a continuous function $\mathbb{R}^d \to \mathbb{R}$ with support $(0,1)^d$.

2.1 1-Dimensional Tree Structure

In one dimension ($d = 1$) the hierarchical structure can be described by an annotated complete binary tree, as exemplified in Figure 1. Each node v is

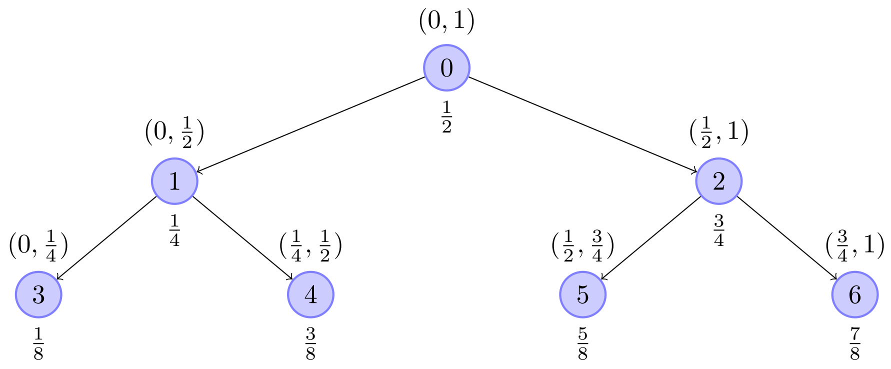

Figure 1: **The 1-dimensional sparse grid of level 2** The tree T_l, the associated intervals above and the centerpoints below. The BFS-number is given inside the circle of the nodes.

annotated with an interval $I_v = (a_v, b_v) \subseteq (0,1)$ leading to the centerpoint $c_v = \frac{a_v+b_v}{2}$. The root is annotated with the open interval (0,1) and the centerpoint 1/2. Each node v of the tree has two children l and r that are annotated with the intervals $I_l = (a_v, c_v)$ and $I_r = (c_v, b_v)$. Note that the centerpoints are unique and can be used to identify the node and the interval. Note further that two intervals of nodes u and v in the tree are either disjoint or one is contained in the other. In the latter case, if $I_u \subseteq I_v$, then v is an *ancestor* of u, i.e., the path from u to the root passes through v. For any node v, the endpoints a_v and b_v of its interval are either 0, 1, or a centerpoint of an ancestor of v. These ancestors are called the *hierarchical predecessors* of v. There can be at most two hierarchical predecessors, namely one for a_v and another for b_v. We define the level of the root node to be 0, and the level of a node v, denoted by $l(v)$, to be its distance to the root, i.e., the number of times one has to follow a parent link to reach the root node. At level ℓ of the tree there are 2^ℓ nodes, all associated intervals have length $b_v - a_v = 2^{-\ell}$ and they partition the interval $(0,1)$ (ignoring that centerpoints of nodes with level $< \ell$ are not element of any interval). We call the tree up to and including level ℓ the *ℓ-tree*, denoted by T_ℓ. Performing an in-order traversal of T_ℓ and collecting the centerpoints yields an equidistant grid in the interval $[0,1]$ with spacing $2^{-(\ell+1)}$ and $2^{\ell+1} - 1$ points.

For every node v of an ℓ-tree we define a basis element of the one-dimensional sparse grid space. In this work, the one dimensional basis function f_v of a node v is piece-wise linear hat function with support (a_v, b_v) and the maximum of 1 at the centerpoint c_v. Observe that the nodes u with $f_u(c_v) > 0$ are precisely the ancestors of v.

A function f in the one-dimensional sparse grid space of level ℓ is given by coefficients λ_v, one for each node of the ℓ-tree, i.e.

$$f = \sum_{v \in T_\ell} \lambda_v f_v \,.$$

Such a function f is continuous and piece-wise linear on $[0,1]$ with kinks at the centerpoints of the nodes of T_ℓ and support $(0,1)$, and it is an element of the one-dimensional sparse grid space. Note that the value $f(c_v)$ at a centerpoint c_v is in general different from the coefficient λ_v.

Definition 1 (The Task of 1-D Hierarchization)

Input *Values y_v, one for each node $v \in T_\ell$.*

Output *Coefficients λ_v such that the represented function $f = \sum \lambda_v f_v$ has the property $f(c_v) = y_v$ for each node v of the ℓ-tree.*

The coefficients λ_v are also called *hierarchical surpluses.*

Algorithm 1: 1-D hierarchization

```
Input  : values at the grid points, stored in $y[\,]$
Output: hierarchical surpluses, stored in $\lambda[\,]$
for $i = maxlevel$ downto $0$ do
    foreach node $v$ of $T_l$ with $l(v) = i$ do
        Let $l_v$ be the left hierarchical predecessor of v
        Let $r_v$ be the right hierarchical predecessor of v
        $\lambda[v] = y[v] - 0.5 * (y[l_v] + y[r_v])$
```

One dimensional hierarchization can be achieved by the pseudocode given in Algorithm 1. To see why, we argue that $y_v = \frac{y_l + y_r}{2} + \lambda_v$ holds. Consider any node $v \in T_\ell$. Observe that for all $u \in T_\ell, u \neq v$ the basis functions f_u falls in one of the following two cases. The first case is $f_u(a_v) = f_u(c_v) = f_u(b_v) = 0$, either because the support of f_u and f_v does not overlap, or because u is a descendant of v, i.e., in u is in the subtree below v. The second case is that f_u is linear in the complete interval $[a_v, b_v]$, which means u is an ancestor of v. Hence, the contribution of all other functions together will result in the linear interpolation between y_l at a_v and y_r at b_v, leading to the equation $\lambda_v = y_v - \frac{y_l + y_r}{2}$.

2.2 Higher Dimensional Case

In higher dimensions ($d > 1$), the sparse grid space is constructed using a tensor product approach, and the term sparse becomes meaningful.

We generalize the index set T_ℓ to its d-fold Cartesian product $T_\ell^d = T_\ell \times \cdots \times T_\ell$. For a vector of d tree nodes $\mathbf{v} = (v_1, \ldots, v_d) \in T_\ell^d$ the level of $\mathbf{v}$ is defined as the sum of the levels $\ell(\mathbf{v}) = \sum_{i=1}^d l(v_i)$. We define its d-dimensional basis function by $f_{\mathbf{v}}(x_1, \ldots, x_d) = \prod_{i=1}^d f_{v_i}(x_i)$. The support of $f_{\mathbf{v}}$ is $I_{v_1} \times$

$\cdots \times I_{v_d}$ and its unique centerpoint $c_{\mathbf{v}} = (c_{v_1}, \dots, c_{v_d}) \in (0,1)^d$ is called a grid point. Note that the d-dimensional volume of the support of $f_{\mathbf{v}}$ is $2^{-\ell(\mathbf{v})}$. The sparse grid of dimension d and level ℓ is based on the set of *sparse grid vectors* $I^d_\ell = \{\mathbf{v} \in T^d_\ell \mid \ell(\mathbf{v}) \leq \ell\}$, and the *sparse grid space* is the span of the corresponding basis functions $F^d_\ell = \{f_{\mathbf{v}} \mid \mathbf{v} \in I^d_\ell\}$. The set of *sparse grid points* is $C^d_\ell = \{c_{\mathbf{v}} \mid \mathbf{v} \in I^d_\ell\}$.

Note that in most of the established literature, the level of the one dimensional functions is counted starting from 1 for what we call the root node. This leads to the situation that the simplest d-dimensional basis function with support $(0,1)^d$ has level d, whereas in our notation it has level 0. In other words, what we call level represents directly the number of refinement steps and hence the volume of the support, independently of the dimension of the underlying space.

With these definitions, the one dimensional hierarchization task formulated in Definition 1 naturally generalizes to higher dimensions.

Definition 2 (The Task of Hierarchization)

Input *Values $y_{\mathbf{v}}$, one for each sparse grid vector $\mathbf{v} \in I^d_\ell$*

Output *Coefficients $\lambda_{\mathbf{v}}$ such that the represented function $f = \sum_{\mathbf{v} \in I^d_\ell} \lambda_{\mathbf{v}} f_{\mathbf{v}}$ has the property $f(c_{\mathbf{v}}) = y_{\mathbf{v}}$ for each $\mathbf{v} \in I^d_\ell$.*

Its algorithmic solution will be the focus of this work. The well established algorithm follows the unidirectional principle [3], making use of the one dimensional algorithm. A *pole* of the sparse grid in direction i containing the sparse grid vector $\mathbf{v} = (v_1, \dots, v_d)$ is the subset of sparse grid vectors that differ from $\mathbf{v}$ only in dimension i. All such poles have the structure of a one dimensional sparse grid (by projection to dimension i), even if some consist of only a single element. Further, the poles in direction i partition the sparse grid. The hierarchization algorithm considers the dimensions one after the other, i.e., it works in the directions $i = 1, \dots, d$. In each iteration it runs Algorithm 1 on all poles of direction i (one-dimensional hierarchization). The pseudocode of this well

Algorithm 2: High Dimensional Hierarchization

```
for $i = d$ downto 1 do
    foreach pole $p$ of the sparse grid in dimension $i$ do
        perform one-dimensional hierarchization on $p$
```

established solution is given as Algorithm 2 and we do not modify it at this level of abstraction. Beyond this pseudocode, several important aspect need to be addressed:

- Which computation happens when.
- How and where the variables are stored.
- How the data is moved.

3 Memory Efficient Algorithm

This section describes the ideas that lead to a memory efficient algorithm for the hierarchization of a sparse grid of high dimension and low level.

3.1 Data Layout

At the heart of the algorithm is a specific layout of the data. From this layout the algorithm itself follows naturally.

3.1.1 Layout in 1 dimension

Given the tree structure detailed in Section 2.1, a breadth-first-search (BFS) traversal of the nodes is well defined: It starts at the root, and then traverses the nodes of the tree with increasing level and from left to right. More precisely, we assign the BFS-number 0 to the root, BFS-number 1 to its left child, 2 to its right child, the BFS-numbers 3, 4, 5, 6 to the four nodes of level 2, and so on. These BFS-numbers are shown in Figure 1. The BFS-number is a unique identifier of a node in the one dimensional tree T_ℓ. This BFS-layout for a complete binary tree is well understood (its perhaps most prominent use is in the heap-sort algorithm). For a node with BFS-number i, its two children have the BFS-number $2(i+1)-1$ and $2(i+1)$, and its parent has BFS-number $\lfloor (i-1)/2 \rfloor$. This simplicity of computing related BFS-numbers is one of the big advantages of the BFS-layout. Note that starting the counting at 1 would make this even simpler, but indexing the array in C-style starting from 0 is not only closer to the code, but it is also more natural when working with subarrays. Observe that the BFS-number encodes the level and the index within the level in one number. The BFS-numbers of nodes with level ℓ start with $2^{\ell+1}-1$. We define the level of a BFS-number to be the level of the corresponding node in the ℓ-tree. Further, a tree of maximum level ℓ uses precisely the BFS-numbers in $\{0,\ldots,2^{\ell+1}-2\}$. We use the BFS-number as a position in the layout. A one dimensional sparse grid (i.e. a pole) can in this way be represented as an array (with one entry per degree of freedom).

3.1.2 Higher Dimensional Layout

Following the tensor product structure of the sparse grid, it is natural to identify a sparse grid point (defined by a vector of tree-nodes $\mathbf{v} = (v_1,\ldots,v_d)$) with a vector $\mathbf{b} \in \mathbb{N}_0^d$ of BFS-numbers. Each such vector of a sparse grid of level n has $\sum_{i=1}^{d} \ell(b_i) \leq n$. We sort the vectors of BFS-numbers lexicographically, with the significance of the positions decreasing (as is usual). This directly leads to a layout of a higher dimensional sparse grid, as exemplified in Figure 2 in 2 dimensions and level 2. Because the first position is most significant, all elements with a 0 at the first position are grouped together at the beginning, then come the elements with a 1 at the first position and so on. Within these groups the sorting bundles together the elements with the same entry at the second position. And within these groups, the elements with the same entry at the third position are grouped together, and so on.

At the end of this conceptual recursion we see groups of vectors that differ only in the entry at the last position and are sorted by this entry. Hence, we see that the poles (defined in Section 2.2) in dimension d form subarrays in the complete lexicographical layout.

0:$(\frac{1}{2},\frac{1}{2})$ 1:$(\frac{1}{2},\frac{1}{4})$ 2:$(\frac{1}{2},\frac{3}{4})$ 3:$(\frac{1}{2},\frac{1}{8})$ 4:$(\frac{1}{2},\frac{3}{8})$ 5:$(\frac{1}{2},\frac{5}{8})$ 6:$(\frac{1}{2},\frac{7}{8})$

7:$(\frac{1}{4},\frac{1}{2})$ 8:$(\frac{1}{4},\frac{1}{4})$ 9:$(\frac{1}{4},\frac{3}{4})$

10:$(\frac{3}{4},\frac{1}{2})$ 11:$(\frac{3}{4},\frac{1}{4})$ 12:$(\frac{3}{4},\frac{3}{4})$

13:$(\frac{1}{8},\frac{1}{2})$

14:$(\frac{3}{8},\frac{1}{2})$

15:$(\frac{5}{8},\frac{1}{2})$

16:$(\frac{7}{8},\frac{1}{2})$

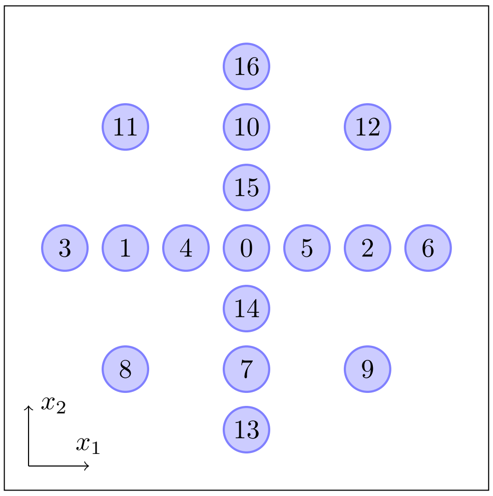

Figure 2: **The 2-dimensional sparse grid of level 2:** On the right the grid points at their positions in $[0,1]^2$, labeled with their position in the layout. On the left the lexicographical layout of the grid points as triples *layout-position*:(x_1, x_2). Note the matrix structure of both figures: rows have the same x_1 coordinate, columns the same x_2 coordinate. The restriction on the level-sum yields a symmetric shape.

3.2 Numerical Work: Hierarchize All Poles

The high dimensional hierarchization Algorithm 2 works pole-local in the inner loop. In other words, when considering dimension i, there is no interaction between variables of different poles in dimension i. Let us focus, for the moment, on the first iteration of the outer loop of Algorithm 2, where all poles of dimension d are hierarchized. This task fits particularly well to our layout of the sparse grid. Each such pole is located in a subarray, starts at a certain offset and has length $2^\ell - 1$ for a pole of level ℓ. Note that some of these poles have level 0, i.e. consist of a single element and hence require no numerical work at all.

Consider Algorithm 1 when it operates on a pole stored in BFS-layout. More precisely, consider the j-th input pole as stored in a vector $\mathbf{y}_j$ and the output in a vector $\mathbf{h}_j$, both with $N = 2^\ell - 1$ entries. We express Algorithm 1 as $\mathbf{h}_j = H_\ell \cdot \mathbf{y}_j$ for a sparse matrix H_ℓ. Consider a node v of the ℓ-tree. The variables of the input and output associated with this node are stored in the same position in the two vectors, say at position i. Hence, on the diagonal, all entries are 1. The variables associated with the hierarchical predecessors are stored before i. Hence, the matrix H_ℓ has at most two non-zero entries below the diagonal, and each of them has value $-1/2$. Note that H_k is the upper left corner of H_ℓ for $k < \ell$.

We can also express the whole first iteration of Algorithm 2 as a matrix multiplication. To this end consider all input values stored in the vector $\mathbf{y}'$ and the result in the vector $\mathbf{h}'$, both according to our layout. Let us describe the matrix W_d for which we have $\mathbf{h}' = W_d\mathbf{y}'$. The matrix W_d is a block diagonal matrix composed of many matrices H_{l_j}, where l_j is the level corresponding to the size of pole j. Hence, on the diagonal of W_d all entries are 1, above the diagonal all entries are 0, there are at most two entries of value -1/2 in every row and they are at distance at most 2^ℓ below the diagonal for a sparse grid of level ℓ.

By the above discussion it is clear that also the other hierarchization steps for dimension $i \neq d$ are linear, but the structure of the corresponding matrices would be different and harder to describe. Hence, in the following, we use a rotation to reuse W_d.

3.3 Rotation

We achieve the outermost loop of Algorithm 2 by performing rotations. In this way the one dimensional algorithm formulated as W_d in Section 3.2 can operate on all dimensions in turn.

Consider the shift $S(n_1, \dots, n_d) = (n_2, n_3, \dots, n_d, n_1)$ working on d-dimensional BFS-vectors. When following the movement of the corresponding centerpoints when applying S to each grid point, geometrically we see a rotation operating on $[0,1]^d$. Because we are working with a non-adaptive sparse grid, this grid-point exists in our sparse grid as well. Observe that S^d is the identity, and that S^i maps the $(d-i)$-th position of the vector to the last position. In terms of our algorithm, using this rotation means that we should take a variable of our vector, understand it as a grid-point/BFS-vector, rotate it with S, and move it to the position in the vector associated with the rotated grid-point. We expresses this data movement by a permutation matrix R.

With this definition of R, we can express one execution of the outer loop of Algorithm 2 as applying the matrix RW_d, and the complete algorithm as applying $(RW_d)^d$. In other words, to transform a vector $\mathbf{y}$ of function values at the grid-points to a vector of hierarchical surpluses $\mathbf{h}$ (both in our layout), we can use the equation

$$\mathbf{h} = (RW_d)^d\mathbf{y}$$

to express Algorithm 2.

3.4 Considerations of an Implementation

With this description of the algorithm as the alternating application of two sparse matrices, it is quite natural to work with two vectors as in the pseudocode given in Algorithm 3.

This code obviously relies on the definition of the matrices W_d and R in a crucial way, and these matrices are based on the correspondence between positions in the layout, BFS-vectors, and the tree structure in the BFS-numbers. We call the translation of a BFS-vector to the position in the layout `pos`, and the reverse operation `depos`. They can be implemented quite efficiently in $O(\ell + d)$, but by the nature of having a BFS-vector with d entries as input or output, they

Algorithm 3: Hierarchization as Sparse Matrix Multiplication

```
Input  : values at the grid points, stored in vector $y[\,]$
Output: hierarchical surpluses, stored in vector $y[\,]$
Let $x$ be a vector of same size as $y$
for $i = 1$ to $d$ do
    $x = W_d * y$
    $y = R * x$
```

cannot take constant time. This can be avoided by considering the BFS-vector as a data structure that changes its state according to well defined operations, in particular changing the BFS-vector such that the corresponding position in the layout is incremented. More precisely, we have an abstract data type *BFS-vector* that stores as it state a $\mathbf{b} = (b_1, \ldots, b_d)$ and supports the following operations:

`int pos():` Return $p(\mathbf{b})$, the position in the layout of $\mathbf{b}$.

`init():` set $\mathbf{b} = (0, \ldots, 0)$.

`increment():` Change $\mathbf{b}$ such that `pos` is incremented, i.e., for the current state $\mathbf{b}$ and the new state $\mathbf{b}'$ it holds $p(\mathbf{b}') = p(\mathbf{b}) + 1$.

`int shift_pos():` Return $p(S(\mathbf{b}))$, i.e. the position of the shifted current BFS-vector.

`int last_entry():` Return b_d.

`depos(x):` Change the current vector such that $p(\mathbf{b}) = x$.

All of the above operations but `depos` can be implemented in $O(1)$ time using a sparse representation of the BFS-vector, as detailed in Section 4.

With this presentation of the hierarchization algorithm, there are several design choices to be taken. Either we can implement a carefully tuned routine that has the effect of applying W_d and R, or we can explicitly represent the two sparse matrices. Which alternative is superior depends on the circumstances and the used hardware: If several hierarchization tasks are performed, the dimension is high, and the access to the sparse matrices is fast enough, the time for precomputation of R and W_d can be amortized by sufficiently many multiplications with the matrices. If instead bandwidth to the memory is much more precious than computation on the CPU, then an on the fly computation is preferable.

Another important concern is the possibility to turn the above algorithm into a parallel one. The multiplication steps are trivial to parallelize including the load balancing. Also the creation of the matrices can easily be parallelized. Only the initialization of the individual parallel threads needs some care and uses `depos`.

4 Navigating the High Dimensional Layout

This section is concerned with computing the position of a BFS-vector $\mathbf{v} = (v_1, \ldots, v_d)$ in the layout described in Section 3.1.2. It relies on knowing the size of a sparse grid of dimension d' and level ℓ' for all $d' \leq d$ and $n' \leq n$.

4.1 The Concept of Remaining Level

In this section we work exclusively with BFS-vectors with dimension d and level $\ell \le n$ to represent the points of the sparse grid. Remember that we use $l(b_i)$ to denote the level of the i-th digit, and that a BFS-vectors $\mathbf{b} = (b_1, \dots, b_d)$ belongs to the sparse grid if and only if $\sum_{i=1}^{d} l(b_i) \le n$.

Assuming that the first k entries of $\mathbf{b}$ are fixed to $b_1, \dots, b_k$, we want to know which values are still possible for $b_{k+1}, \dots, b_d$. This is restricted by the limit on the sum of the levels of the complete vector. Let the *remaining level* be defined as $r = n - \sum_{i=1}^{k} l(b_i)$. If $r < 0$, then the initial entries do not lead to a grid point, regardless of the remaining entries. Otherwise, the level sum of the remaining entries has to be bounded by r for the complete BFS-vector to belong to the sparse grid.

4.2 The number of Grid Points $v(d, \ell)$

One important quantity is $v(d, \ell)$, the number of grid points (size) of a d-dimensional sparse grid of level ℓ, or equivalently, the number of d-dimensional BFS-vectors with level sum bounded by ℓ. In the following, we use a recursive formula for v.

For the base case, it is convenient to understand a 0-dimensional sparse grid to consist of precisely one point, regardless of the level. Hence,

$$v(0, \ell) = 1 \qquad \text{for all } \ell \in \mathbb{N}_0 \,. \tag{1}$$

The recursive case is given by

$$v(d, \ell) = \sum_{i=0}^{\ell} 2^i \cdot v(d-1, \ell - i) \,. \tag{2}$$

Here the quantity 2^i represents the number of grid points in a BFS-tree with level precisely i.

The validity of (2) follows, for example, from the layout proposed in Section 3.1.2: The first group has a 0 in the first position, the remaining level is ℓ, and the group is a sparse grid with level ℓ in $d-1$ dimensions. The next 2 groups have in the first digit a value of level 1, which means that the remaining level is $r = \ell - 1$ and each of them is a grid with dimension $d-1$ and level r. This argument continues with increasing level of the first digit, finally reaching the digits of level ℓ. These are 2^ℓ many groups, each with a remaining level 0. This means that the remaining digits must all be 0, which is consistent with our definition that a level 0 grid consists of a single grid point.

The formulas (1) and (2) form a recursion because the right hand side depends only on cases with smaller or equal level and one dimension less. For efficiency we usually compute the (relatively small) table for $v(d', n')$ using dynamic programming.

4.3 Operations on a Full Vector

4.3.1 `pos`: the Position in the Layout

With the help of $v(d, \ell)$ we can compute for a BFS-vector $\mathbf{b} = (b_1, \dots, b_d)$ the position in the layout $p(\mathbf{b})$. This position can be computed by Algorihm 4,

Algorithm 4: `pos`(), the position of a BFS-vector in the layout

```
Input  : the BFS-vector in $b[\,]$
Output: the position in $p$
$p = 0$
$l =$ maxlevel
for $i = 1..d$ do
    for $x = 0..b[i]-1$ do
        $p = p + v(d-i, l - \texttt{level}(x))$
    $l = l - \texttt{level}(b[i])$;
```

where `level`(x) denotes the level of the 1-D-tree node with BFS-number x, and `maxlevel` denotes the overall level ℓ of the sparse grid, and the BFS-vector is stored in the array $b[\,]$ (indexed with $1, \ldots, d$).

We can think of the addition to p as jumping in the layout. To account for the first entry $b_1 \in \{0, \ldots, 2^{\ell+1} - 2\}$, we jump to the group of BFS-vectors with first entry b_1, namely to the position of the BFS-vector $(b_1, 0, \ldots, 0)$. From this we can continue in the same manner for the remaining vector $(b_2, \ldots, b_d)$, interpreting this vector as an element of the sparse grid with $d-1$ dimensions and the remaining level. Note that for the last entry the increments are by $1 = v(0, \ell)$.

4.3.2 `depos`: the BFS-vector from a Position

Now we consider the operation `depos`, the inverse to `pos`, as formulated in Algorithm 5. Given a position p, we want to find the BFS-vector $\mathbf{b} = (b_1, \ldots, b_d)$ that is mapped to p in the layout. The structure of `depos` is very similar to `pos`. Instead of jumping to the starting position of a group, we want to find the group in which p is located. In other words, the algorithm consists of d linear

Algorithm 5: `depos`(), the BFS-vector from the position in the layout

```
Input  : the position in $p$
Output: the BFS-vector in $b[\,]$
freelevel=maxlevel
$b = (0, .., 0)$
for $i = 1..d$ do
    $w = v(\texttt{dim} - i, \texttt{freelevel})$
    while $p >= w$ do
        $p = p - w$
        $b[i] = b[i] + 1$
        $w = v(\texttt{dim} - i, \texttt{freelevel} - \texttt{level}(b[i]))$
    $\texttt{freelevel} = \texttt{freelevel} - \texttt{level}(b[i])$
```

searches for the group of dimension i (inside the group of dimension $i-1$) in which p is located.

4.3.3 `increment`: the Next BFS-vector in the Layout

The functions `pos` and `depos` as explained in the last two Sections are not efficient enough to be used in the innermost loop of a hierarchization algorithm, a problem we circumvent by an `increment` operation. We store the BFS-vector $\mathbf{b} = (b_1, \dots, b_d)$ in the array $b[\,]$, and the corresponding position p in the layout in the variable p. The `increment` operation has the purpose of incrementing p and changing $\mathbf{b}$ in a way that preserves this correspondence. This and the direct access to b_d are sufficient to create W_d (or to compute $y = W_d \cdot x$ without a representation of the matrix) serially. For a parallel version of this algorithm we assign each thread the task to create a range $p_i : p_{i+1} - 1$ of rows of W_d. Then the thread starts by setting $p = p_i$ and establishing the invariant by initializing the vector using $\texttt{depos}(p_i)$. From then on, `depos` is no longer needed.

Assume for the moment that there is an additional array of the remaining levels $r[k] = \ell - \sum_{i=1}^{k} l(b_i)$ as defined in Section 4.1. If $r[d] > 0$, the current vector does not exhaust the maximum level, and the last entry $b[d]$ can be incremented without violating the constraint on the level sum. Even if $r[d] = 0$, it is possible that $b[d]$ is not the highest BFS-number of level $r[d-1]$, and hence the increment changes only $b[d] = b[d] + 1$.

If $b[d]$ is the the highest BFS-number of level $r[d-1]$, we know that the increment has to change some of the entries further to the left, and the incremented vector has $b[d] = 0$. The change further left is an increment operation on the $d-1$ dimensional BFS-vector stored in $b[1], \dots, b[d-1]$ with level-sum ℓ, and we can continue in the same way. If this process would continue to the left of $b[1]$, this means that the first entry $b[1]$ is the highest allowed BFS-number (and all other digits are 0). Then an increment is impossible because we reached the last point of the sparse grid.

After updating the digits $b[\,]$ as described (from right to left), we also update all $r[\,]$ values that might have been changed (from left to right). Note that it is actually not necessary to work with a complete vector of $r[\,]$-values. Instead, we can have a single variable r (`freelevel` in the code) that maintains $r[d]$, i.e., by how much the current vector does not exhaust the level ℓ.

4.3.4 `shift_pos`: the Position of the Shifted BFS-vector

For the computation of R we can use `increment`, if we manage to update the position of the shifted BFS-vector efficiently. More precisely, if our current array $b[\,]$ represents the BFS-vector $\mathbf{b} = (b_1, \dots, b_d)$, we want to compute $\texttt{pos}(S(\mathbf{b}))$. We use some additional arrays that need to be updated during `increment`.

An increment operation leaves a prefix $b_1, \dots, b_k$ of $\mathbf{b}$ unchanged, which means that the prefix $b_2, \dots, b_k$ of length $k-1$ of $S(\mathbf{b})$ remains unchanged. The algorithm of `pos` (Section 4.3.1) considers the entries of the BFS-vector from left to right. Hence, an unchanged prefix of the BFS-vector means that the initial operations of `pos` are unchanged. This can be used by recording the state of the algorithm for $\texttt{pos}(S(\mathbf{b}))$ at each entry into the body of the loops. More precisely, this gives two arrays, one for the position p and one for the remaining level l, both two-dimensional, indexed by i and x. It is sufficient for these arrays to be current in each dimension i up to the position $b[i]$.

Every increment operation increments one entry, say from b_k to $b'_k = b_k + 1$, and all entries to the left are zero, i.e., $b'_{k+1} = \cdots = b'_d = 0$. Hence, we can "jumpstart" the execution of $\texttt{pos}(S(\mathbf{b}))$ starting from what we find as state in $[k-1, b_k]$, the offset implementing the shift. From there we can complete (including recording) the execution of `pos` on the entries $(b_{k+1}, \ldots, b_d, b_1)$. Observe that the control-flow of this remaining execution is always the following: There is the last iteration of the inner loop with $i = k-1$ and $x = b_k$ (creating a new entry in our arrays). For $i = k, \ldots, d-1$ we only update our tables at $[i, 0]$ recording the unchanged state of position p and remaining level r. The inner loop is not executed. Only for $i = d$ the inner loop iterates up to b_1, which always results in adding b_1 to p. Hence, the running time of this continuation of `pos` is proportional to the number of entries consider by `increment`, namely $d - k + 1$.

4.3.5 Performance Limitations of the Full Representation

It is well known that counting with a binary representation of the number changes amortized constant many bits per increment operation. In analogy, we might hope for a good performance of the operations `increment` and `shift_pos` in the amortized sense. This is actually not the case. In the following example the amortized cost per `increment` operation is lower bounded by $\Omega(d)$. Consider the case of level 2 and a very high dimension d. Then, almost all BFS-vectors have precisely two ones, and there are $\binom{d}{2}$ such vectors. For such a vector the cost of an increment is given by the distance of the last 1 to the right. Now all vectors that have at least $d/2$ trailing zeros require $d/2$ work, and there are roughly $\binom{d/2}{2}$ of them, i.e., roughly a fourth of all vectors. In this case enumerating all BFS-vectors with an `increment` operation takes time cubic in d, which means that the amortized time per increment is $\Omega(d)$.

4.4 Sparse Implementation

As detailed in the previous Section, the increment operation on the full BFS-vector becomes problematic if there are many trailing zeros. This immediately suggests to use a sparse representation of the vector. This means storing a list of pairs (dimension, BFS-number), one for each entry that differs from zero, sorted by dimension.

Observe that the number of pairs is not only at most the dimension d, but also at most the level n of the sparse grid.

With this sparse representation, the increment takes only $O(1)$ operations, as we will argue in the following. The algorithm has to distinguish between several cases as listed below. Observe that for each case the sparse representation allows to test if it is applicable, and if so, to update the representation in constantly many operations, including the remaining level of the current BFS-vector.

- If the remaining level of the current vector is > 0, the last entry b_d can be incremented. This might extends the list by the pair $(d, 1)$.
  Hence, in all other cases we can assume that the current vector has remaining level zero.

- If the last non-zero entry can be incremented without changing the level, we do so.

- If the list consists of the single entry $(1, 2^{n+1} - 2)$ the increment operation is impossible (last BFS-vector).
- If the rightmost non-zero entry is b_i (in dimension $i > 1$ and it cannot be incremented), and $b_{i-1} = 0$, we set $b_i = 0$ and $b_{i-1} = 1$.
- Otherwise the rightmost two non-zero entries are b_{i-1} and b_i. In this case we set $b_i = 0$ and increment b_{i-1}. This is always possible because changing b_i to 0 decreases the level by at least one, whereas incrementing b_{i-1} increases the level by at most one.

It is easy to adjust the computation of `depos`, `pos` and its incremental version needed for `shift_pos` to the sparse representation. Overall this means that all operations, but `depos`, of the abstract data type formulated in Section 3.4 can be implemented in worst case constant time.

5 Implementation and Experiments

The described algorithms are implemented as a proof of concept and we compare running times for the hierarchization tasks with the implementation in SG++. The new approach presented in this paper is abbreviated as `rSG` (rotating Sparse Grid). SG++ is designed to deal with adaptive refinements and cannot take advantage of the simpler structure of a complete sparse grid. Further, as it is currently bundled, it is not parallelized. Another candidate for comparison would be `fastsg`, but its hierarchization procedure is only for sparse grids with boundary points, so it solves a different problem and a comparison would be meaningless. As a proof of concept, the point of the experiments is to show the potential of the new algorithmic ideas. Hence, our focus in these experiments is to show how far we can go beyond the current standard solution. Accordingly, the focus of the results are the sizes of the problems that can be tackled at all and the orders of magnitude of runtime and memory consumption. This is meant to give an indication of the strengths and weaknesses of our new approach.

The current implementation does not (yet) use the constant time algorithm for `shift_pos`. With the current setup of creating the matrices W_d and R once and then applying them several times, the possible savings in runtime are not significant enough.

5.1 Experimental Setup

The implementation is a C++ design that is parallelized with openMP, with a focus on being able to compare the performance of different algorithmic approaches. To this end, templates are used in many situations, whereas inheritance and in particular virtual classes and other constructs that are resolved at runtime are avoided. The experiments are run on a linux workstation with an Intel XEON E3-1240 V2 chip, having 4 cores clocked at 3.4 GHz. The total main memory of the machine is 32Gb. The code is compiled with `gcc 4.4.7` and the compiler flags `-O3 -march=native -fopenmp -D_GLIBCXX_PARALLEL`.

The time measurements generally consider the whole execution time of the algorithm for both SG++ and `rSG`, including the initialization of data structures and setting the input vector with values. This means that in particular the phase where the two matrices W_d and R are computed (preparation phase) includes the time for using the memory for the first time (cold cache).

The time is measured inside the programm using the system call `gettimeofday()`.

The memory requirement is determined by the system utility `time` with the `%M` format, stating "Maximum resident set size of the process during its lifetime."

In this section, we describe the problem instances not only by their dimension and level but also by their number of grid points (DoF).

5.2 Experiments

5.2.1 Parallel Scaling

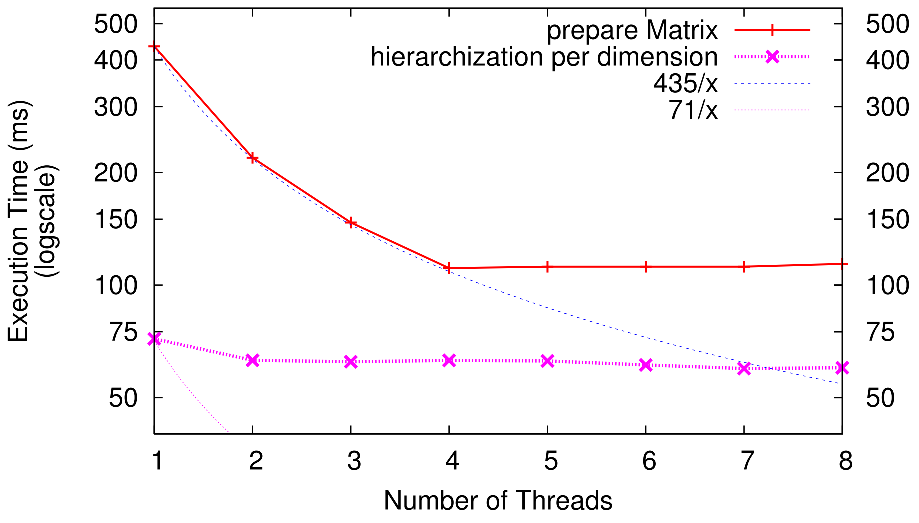

Figure 3: **Strong scaling** behaviour for dimension 20, level 6

To evaluate the effectiveness of the ideas to paralellize the hierarchization algorithm we perform a strong scaling experiment, i.e., we solve the same problem with an increasing number of processors.

As a prototypical example we take the hierarchization of a sparse grid in 20 dimensions and level 6. This sparse grid has roughly 13 million grid points, uses 377 Megabytes to store the grid and the execution uses in total 1.9 Gigabytes memory from the system.

Given that our machine has 4 cores with hyperthreading, we experiment with up to 8 threads. The running times for the different numbers of threads are summarized in Figure 3.

The preparation of the matrices scales perfectly up to 4 threads on the 4 cores, whereas more threads give no further improvement of the running time.

As a third plot in Figure 3, we have the average hierarchization, i.e., the average time for one application of the two sparse matrices. We see that this phase scales poorly, and preparing the matrices takes roughly twice as long as applying them using 4 processors. Hence, with our current implementation of computing the matrices we have an algorithm that scales almost perfectly, but it is in total not fast enough to justify changing to an on the fly computation of the matrices, i.e., not creating a representation of the matrices. With different hardware or a more efficient implementation of the algorithms presented in this paper, this situation might change.

5.2.2 Running times

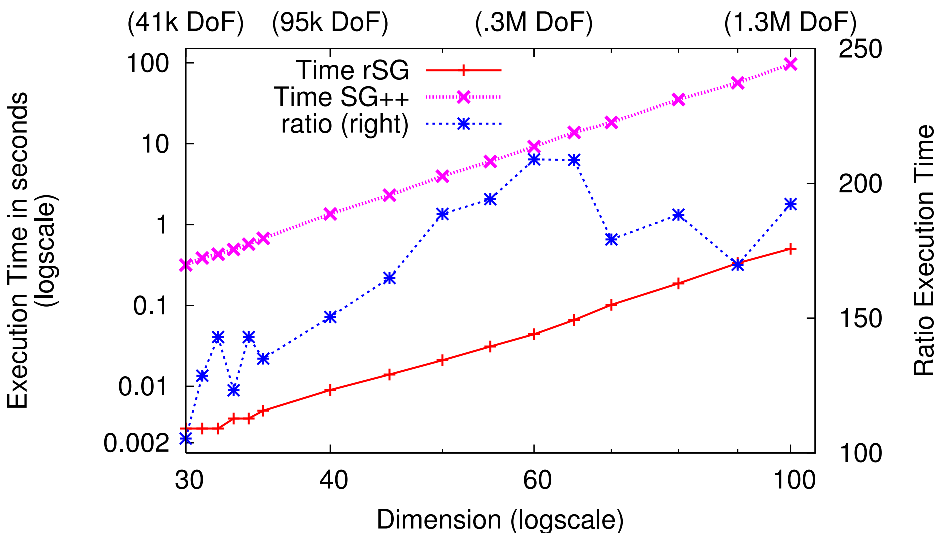

Figure 4: **Serial Run time comparison for different dimensions and level 3:** The x-axis gives the dimension of the sparse grid, labeled at the bottom. The dimension induces a certain DoF, as given above the figure. The y-axis on the left is running time in seconds for the two single core implementations with lines connecting the measurements. The y-axis on the right gives the ratio between the running times as plotted in the blue dashed line.

In the plots given in Figure 4 and 5 we compare the running times for the sparse grids of level 3 and level 6 for different dimensions. The range of dimensions is in both cases chosen in a way that the running times of both implementations are between 1 ms and 3 minutes. In this range the implementation `rSG` (following the ideas presented in this paper) achieves speedups over SG++ as shown in the range of 100-220 for level 3 and respectively 30-53 for level 6.

5.2.3 Memory Footprint

The amount of memory used by a program is not only an important resource in itself, it also has, due to caches, a significant influence on the running time.

In Figure 6, we plot the memory usage of the two implementations `rSG` and SG++ . We see significant differences in the memory usage per DoF of the sparse grid, in particular for high dimensions and high levels. For large instances the `rSG` implementation reaches a space usage of 16 words of 64-bit (i.e. doubles or long integers) per DoF. In contrast, the space usage of SG++ per DoF increases with the dimensionality of the problem.

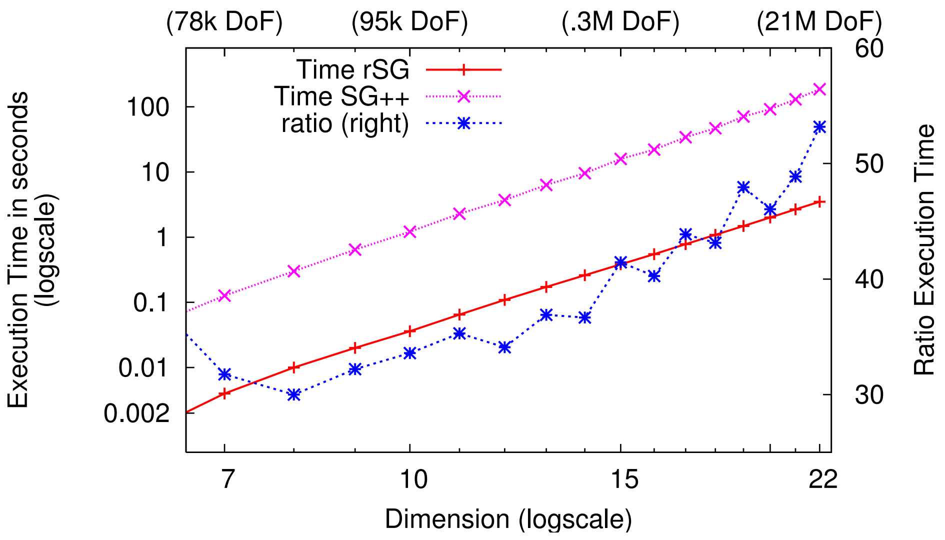

Figure 5: **Serial Run time comparison for different dimensions and level 6:** Same setup as Figure 4.

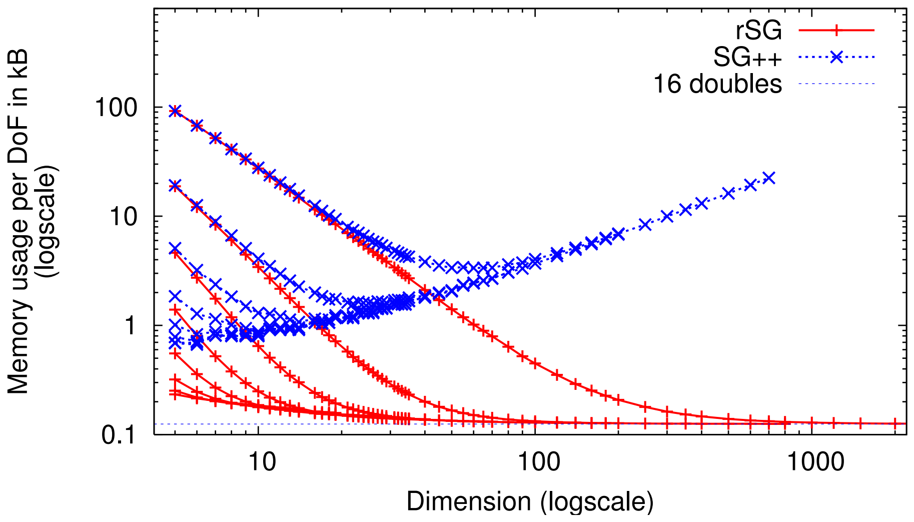

Figure 6: **Memory Usage per DoF:** The y-axis stands for memory usage as determined by `time -f %M` divided by the number of variables (DoF) of the sparse grid. The x-axis stands for the dimension and measurements for the same level are connected. The level-2 line is the topmost where SG++ and `rSG` have the same memory usage for small dimensions.

5.2.4 Solvable Problem Sizes

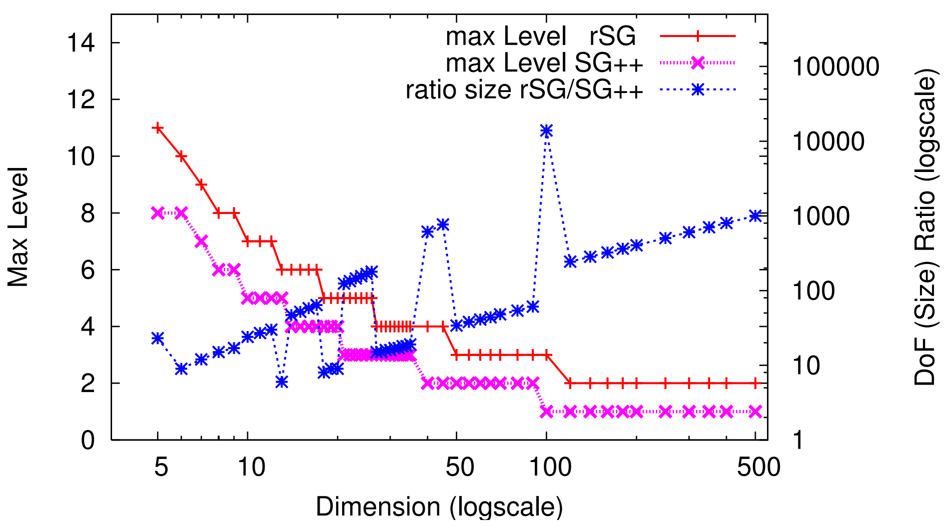

Figure 7: **Maximal sparse grid hierarchization doable serially in one second:** For every sampled dimension, as given on the x-axis, we have the maximum level (left y-axis) that the two single core implements can hierarchize within the resource constraint. In blue the ratio of the DoF of the corresponding (maximal) sparse grids.

Figures 7–9 show the pareto front of problem sizes that can be computed with different resource limitations, namely limited computation time for the serial program and memory consumption. Remember that this comparison is somewhat unfair against SG++ because SG++ does not exploit that the grid is non-adaptive.

The first experiment sets a limit on the computation time for the hierarchization task. Figure 9 limits the used space somewhat arbitrarily to 16GB.

In almost all considered cases, the new approach can hierarchize a sparse grid with at least one more level, sometimes three more levels, than SG++. In particular for high dimensions, the corresponding increase in degrees of freedom is a factor of up to 10'000.

Observe that this comparison is not very fine grained: Increasing the level of the sparse grid by one increases the degrees of freedom by a factor of 10 to 5000. Hence, in many cases in the above comparisons, the resource constraint is by far not exhausted.

5.2.5 Relation between Preprocessing and Hierarchization

We investigate the relation between the preprocessing time to create the matrices W_d and R, and the time to apply them once.

In Figure 10 we see in the serial case that for small grids the preparation time is comparatively expensive. One explanation for this behavior is that the application step becomes memory bound for large grids: As long as the grid is small and everything fits into the cache, we see that the preparation takes a

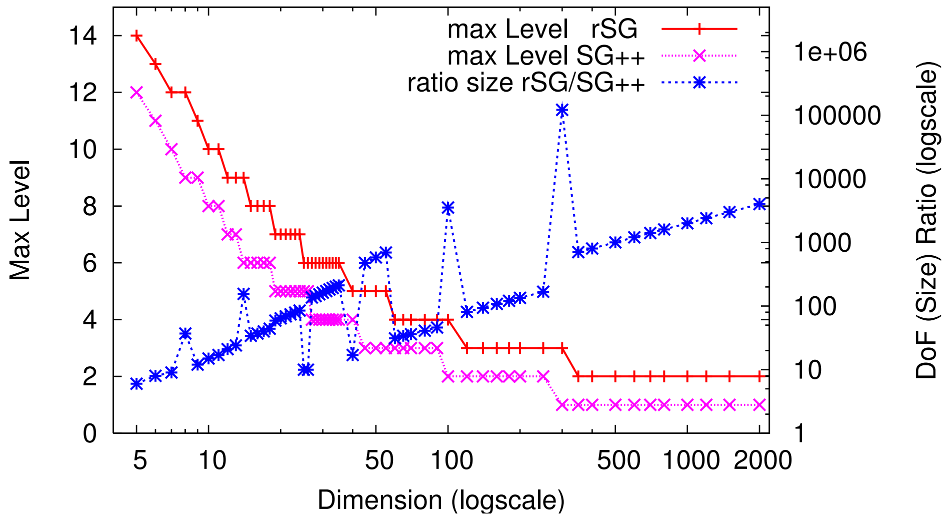

Figure 8: **Maximal sparse grid hierarchization doable serially in one minute:** Same setup as Figure 7.

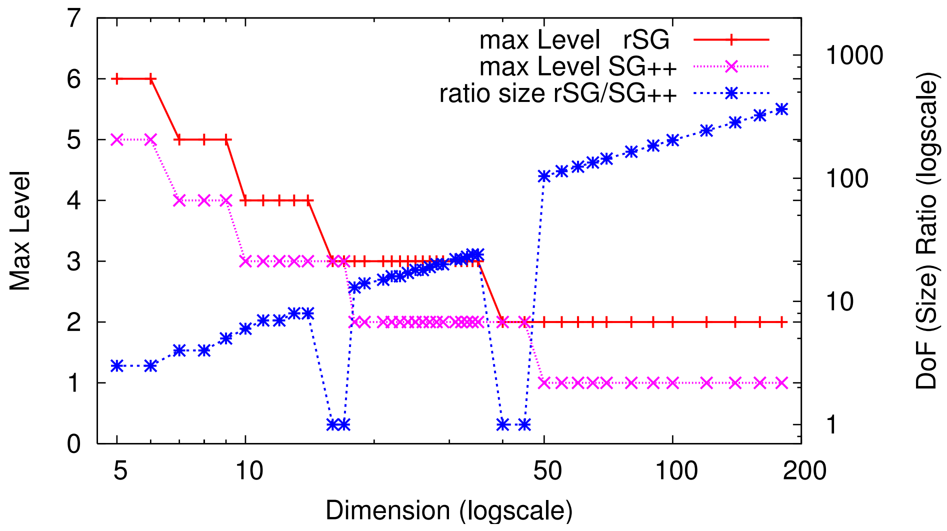

Figure 9: **Maximal sparse grid hierarchization doable with 16GB memory:** For every sampled dimension, as given on the x-axis, we have the maximum level (left y-axis) that the two implements can hierarchize within the resource constraint. In blue the ratio of the DoF of the corresponding (maximal) sparse grids.

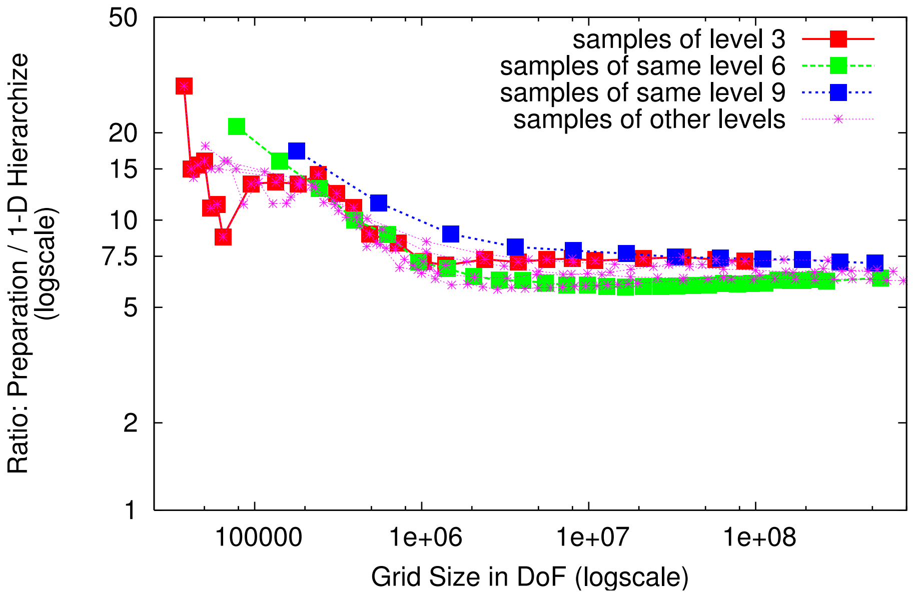

Figure 10: **Preparation vs Hierarchization (serial):** The y-axis shows the factor by which preparing W_d and R is slower than using them. The x-axis gives the size of the sparse grid in DoF. The measurements for the same level are connected by a line.

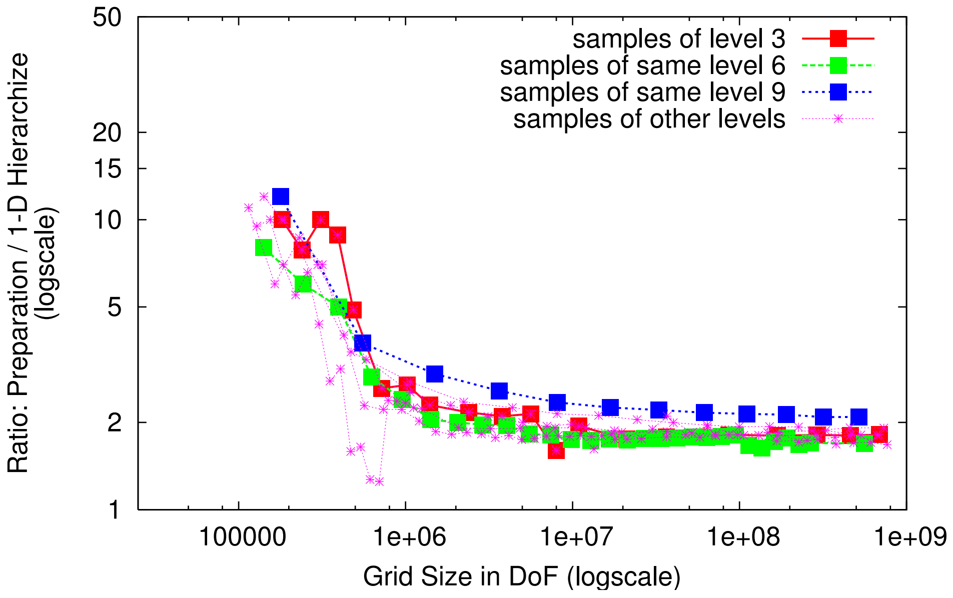

Figure 11: **Preparation vs Hierarchization with 4 Threads:** with the same setup as in Figure 10, and for comparability the same axis. For small grids some points cannot be reported because the measured time is 0.

lot more computing than the application step. As soon as the grid needs to be fetched from main memory because it does not fit into the cache anymore, the application step gets slower, if there are no other effects, by a factor of roughly 3. The preparation phase is spending more time on computations on BFS-vectors, and hence the additional time for the slower memory access accounts for a smaller fraction of the total running time. Perhaps the additional memory latency is even completely hidden because the computation continues while the memory system stores the results.

We repeat this experiment with all four cores in Figure 11. The findings of this experiment give further evidence that the application step is memory bound. It is consistent with the findings of Section 5.2.1 and Figure 3. There we saw that the preprocessing step scales well with additional processors, whereas the hierarchization step improved over the serial case by at most a factor of 1.2, which suggests that the latter one is memory bound. This fits to what we can observe in Figure 11, where we see that the ratio for big sparse grids improves to 1.7–2.

When comparing Figure 10 and Figure 11 directly, one notices that in the parallel case there are fewer samples for small sparse grids. This is due to the fact the computation gets faster, and that these grid are now reported with a 0 running time for hierarchization, and we cannot compute the ratio we are interested in.

Going back to the case of level 6 and dimension 20 considered in Figure 3 (Section 5.2.1) we see that it is indeed prototypical. When locating this measurements in Figure 10 by looking for the level 6 measurements with 1.28e7 DoF, we see that it actually achieves one of the smallest ratios.

One question that can be addressed by the considerations in this section is by how much the implementation of the data type for the BFS-vector needs to improve before an on the fly creation of the matrices is superior. Note that good processor scaling could provide this improvement and that the implementation does not yet use the constant time algorithm for `shift_pos`.

6 Conclusion

In this article, we have been exploring the basics of a new algorithmic idea to work with sparse grids of low level and high dimensions. The main concepts of a compact layout, rearranging the data and generalized counting present themselves as promising ideas. There are many directions in which the topic should be developed further. One direction is to consider vectorized parallel implementation, as done in [5]. Another direction is the evaluation algorithm on the data layout proposed here, as done in [6]. More directly, there is the question if a carefully tuned on-the-fly computation of the two sparse matrices can be beneficial in a parallel setting.

7 Acknowledgments

I am grateful to Dirk Pflüger for introducing me to the topic and providing me with the SG++ source code. For many fruitful discussions I want to thank Gerrit Buse, Dirk Pflüger and Hans Bungartz. Special thanks go to Philipp

Hupp for many helpful comments and discussions to improve the presentation.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems*, 47:934–962, 2010.

[3] H.-J. Bungartz. *Finite elements of higher order on sparse grids, Habilitationsschrift*. Shaker Verlag, Aachen, München, 1998.

[4] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004. URL `http://dx.doi.org/10.1017/S0962492904000182`.

[5] G. Buse, D. Pflüger, A. F. Murarasu, and R. Jacob. A non-static data layout enhancing parallelism and vectorization in sparse grid algorithms. In M. Bader, H.-J. Bungartz, D. Grigoras, M. Mehl, R.-P. Mundani, and R. Potolea, editors, *ISPDC*, pages 195–202. IEEE Computer Society, 2012. ISBN 978-1-4673-2599-8.

[6] G. Buse, D. Pflüger, and R. Jacob. Efficient pseudorecursive evaluation schemes for non-adaptive sparse grids. *Sparse Grids and Applications (SGA)*, 2013.

[7] A. Maheshwari and N. Zeh. A survey of techniques for designing I/O-efficient algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 36–61. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-00883-5. URL `http://dx.doi.org/10.1007/3-540-36574-5_3`.

[8] A. Murarasu, J. Weidendorfer, G. Buse, D. Butnaru, and D. Pflüger. Compact data structure and parallel alogrithms for the sparse grid technique. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[9] A. F. Murarasu, G. Buse, D. Pflüger, J. Weidendorfer, and A. Bode. Fastsg: A fast routines library for sparse grids. In H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *ICCS*, volume 9 of *Procedia Computer Science*, pages 354–363. Elsevier, 2012.

[10] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, München, Aug. 2010. ISBN 9783868535556. URL `http://www5.in.tum.de/pub/pflueger10spatially.pdf`.

[11] URL `http://www5.in.tum.de/SGpp/`.

[12] C. Zenger. Sparse grids, in parallel algorithms for partial differential equations. *Notes on Numerical Fluid Mechanics*, 31, 1991.