Global Communication Schemes for the Numerical Solution of High-Dimensional PDEs

Philipp Hupp[a,*], Mario Heene[b], Riko Jacob[c], Dirk Pflüger[b]

[a] *Institute of Theoretical Computer Science, ETH Zürich, Universitätstr. 6, 8092 Zürich, Switzerland*
[b] *Institute for Parallel and Distributed Systems, University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany*
[c] *IT University of Copenhagen, Rued Langgaards Vej 7, 2300 København S, Denmark*

Abstract

The numerical treatment of high-dimensional partial differential equations is among the most compute-hungry problems and in urgent need for current and future high-performance computing (HPC) systems. It is thus also facing the grand challenges of exascale computing such as the requirement to reduce global communication. To cope with high dimensionalities we employ a hierarchical discretization scheme, the *sparse grid combination technique*. Based on an extrapolation scheme, the combination technique additionally mitigates the need for global communication: multiple and much smaller problems can be computed independently for each time step, and the global communication shrinks to a reduce/broadcast step in between. Here, we focus on this remaining synchronization step of the combination technique and present two communication schemes designed to either minimize the number of communication rounds or the total communication volume. Experiments on two different supercomputers show that either of the schemes outperforms the other depending on the size of the problem. Furthermore, we present a communication model based on the system's latency and bandwidth and validate the model with the experiments. The model can be used to predict the runtime of the reduce/broadcast step for dimensionalities that are yet out of scope on current supercomputers.

Keywords: Global Communication, Sparse Grid Combination Technique, Communication Model, Communication Performance Analysis, Experimental Evaluation, High-Performance Computing

1. Introduction

In the context of partial differential equations, high-dimensional problems are problems that have more than the classical four dimensions – space plus time. A prominent example stems from plasma fusion. The simulation of hot plasmas in fusion reactors of Tokamak or Stellerator type is either based on a gyrokinetic approach or on the Vlasov-Maxwell equations. 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 fully suffer the "curse of dimensionality", a term going back to Bellman in the 60s [1]. The curse of dimensionality describes the fact that the number of unknowns grows exponentially with the dimensionality. Hence, high-dimensional applications are among the compute-hungry drivers of exascale computing [2]. A discretization with 1024 grid points in one dimension would result in 1024^d grid points in d dimensions. This would be more than 10^{15} unknowns for $d = 5$ and would require more than 8 PetaBytes to only store the values in double precision. Thus, such problems are out of scope for computation. Typical simulations have to restrict themselves to small subsets of the domain. But for global simulations of the international flagship project ITER, a Tokamak-style fusion reactor built in Caderache, France, at least 10^{11} degrees of freedom for about 10^6 time steps would be desirable.

*Corresponding author. Tel.: +41 44 633 70 22

Email addresses: philipp.hupp@inf.ethz.ch (Philipp Hupp), mario.heene@ipvs.uni-stuttgart.de (Mario Heene), rikj@itu.dk (Riko Jacob), dirk.pflueger@ipvs.uni-stuttgart.de (Dirk Pflüger)

Fortunately, hierarchical discretization schemes are of help: Given certain smoothness conditions, a discretization based on sparse grids (SG) – a term coined in [3] for the solution of high-dimensional PDEs – can reduce the number of grid points by 8 orders of magnitude from 10^{15} to 10^7 in the example above for only slightly deteriorated accuracy. Here, we employ the sparse grid combination technique [4]. It lets us solve the original problem formulation for many, but coarse anisotropic discretizations also called component grids [5]. A suitable linear combination of the component grid solutions then retrieves a single solution in the hierarchical sparse grid space. For time-dependent PDEs, these smaller problems can be solved independently for a few time steps and therefore embarrassingly in parallel. The hierarchical approach thus overcomes a central problem of massively parallel computations: The splitting 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 case of plasma physics simulations, we are employing GENE [6] in a current project [7, 8] 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.

Between the time steps, or at least every few time steps [9], 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 global communication as illustrated in Figure 1. This exchange is the main remaining communication/synchronization bottleneck and the focus of this work. The current trend in HPC systems and in hardware development will widen the gap between floating point performance and communication bandwidth. To harness future exascale computers, it will be necessary to develop optimized communication schemes. To be able to theoretically analyze the remaining communication bottleneck of the combination technique, we assume that the communication is performed in one step, and that one single process is responsible for each component grid. While these are simplifying assumptions, they are not unrealistic for large-scale simulations where the memory demand of a single grid exceeds more than half of the available memory on a shared memory compute node. Furthermore, the analysis will serve as a profound baseline for more complex setups.

In previous work [10] we have studied optimal communication strategies for this remaining communication bottleneck in the 2- and 3-dimensional settings. This includes the two strategies *Sparse Grid Reduce* and *Subspace Reduce* which we also discuss in this paper. Furthermore, we have presented first bounds for general dimensionalities.

Here, we extensively extend the preliminary work shown there. First, we extend the full analysis from the fixed 2- or 3-dimensional setting, which is much simpler and easier to analyze, to the higher-dimensional case. Second, we enhanced those communication strategies that can be applied to arbitrary dimensionalities. In particular, we extend the strategies to handle sparse grids with boundary points and/or a minimum level, which is required for real-world settings. Third, a reordering of the increment spaces is given that allows a parallel execution of *Subspace Reduce*. Our analysis shows that *Parallel Subspace Reduce* achieves asymptotically (ignoring factors depending only on d) the optimal makespan communication volume. Fourth, we then extend the experiments to higher dimensions and predict the behavior of the algorithms with the theoretical model. Using the enhanced strategies, we conduct measurements on high-performance computing (HPC) systems for 3 dimensions, the 5-dimensional setting of GENE described above, and an extended 10-dimensional setting. Furthermore, we present a communication model which is well-suited to predict the cost of the communication step. Given performance characteristics of the employed HPC system, this model estimates lower and upper bounds for the runtime of the communication step. The model can even be applied to settings that are yet out of scope due to the high computational demand and to predict their communication feasibility for future HPC platforms.

The rest of the paper is organized as follows: in Section 2 we discuss related work including memory and communication models, in particular for parallel computations, sparse grids and the combination technique. Section 3 then covers the necessary background and notation regarding sparse grids. Section 4 introduces the communication model in detail. The communication algorithms as well as lower bounds are presented in Section 5. The communication costs of the algorithms are analyzed in detail in Section 6. The experimental setup as well as the employed HPC systems are described in Section 7. Thereafter, we present estimations based on the communication model as well as the experimental results in Section 8. Section 9 concludes this work by summarizing the results and pointing out directions of future research.

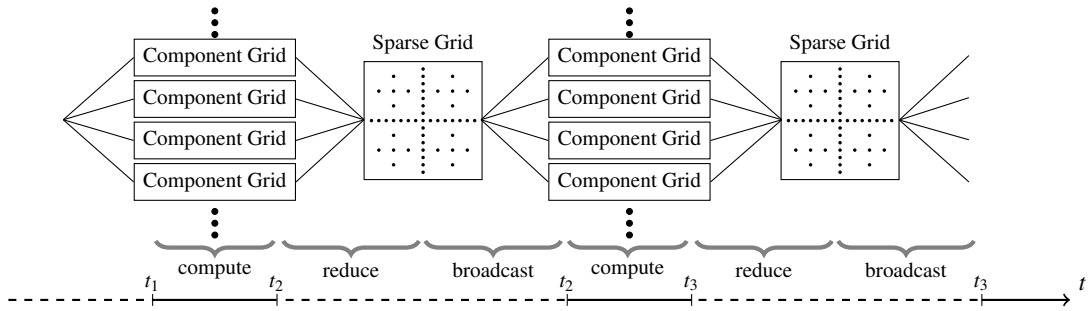

Figure 1: The sparse grid combination technique for a time-dependent problem [9]: For each of the component grids a classical solver is used to compute the next time steps. In the following reduce step the different solutions are accumulated on a sparse grid. Thereafter, the combined solution on the sparse grid is projected and distributed back onto the component grids in a broadcast step.

2. Related Work

2.1. Memory and Communication Models

In many situations it is important to focus not on the CPU operations an algorithm executes, but on the way how data is communicated between different components of the computer. One of the first models to measure the I/O-complexity of algorithms was the *red-blue pebble game* of Hong and Kung [11]. Hong and Kung assume that there is an internal memory of size M in which the data has to reside to compute with it. Single data items (block size $B = 1$) can be moved between internal memory and an infinitely large external memory on which everything resides initially and at the end of an algorithm. The I/O-complexity of the red-blue pebble game is the number of data items that need to be transferred between internal and external memory. Aggarwal and Vitter generalized the red-blue pebble game to the *external memory model* (EM model) [12]. The EM model accounts for spatial locality by transferring data in blocks of size $B \geq 1$ between internal and external memory. There are also various approaches [13, 14, 15] to extend such models to several layers of the memory hierarchy. Arge et al. generalized the EM model to the *parallel external memory model* (PEM model) [16] in which p processors each have their own private cache of size M and 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. The red-blue pebble game and the EM as well as the PEM model have in common that they are cache-aware: the parameters M and 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, Frigo et. al. introduced the cache-oblivious model [17] in which the parameters M and B are not known to the algorithm. The idea is to design algorithms that work efficiently for any M and B and hence efficiently across different machines.

All of the above models assume that local computations are for free and the parallel PEM model assumes that communication between processors has to take place using the external memory. Direct communication between processors is not allowed. The bulk-synchronous parallel model (BSP model) of Valiant [18] takes both into account: direct communication between processors as well as the cost of local computations. 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 when all processors have finished their tasks for the current superstep, they 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 costs of a superstep are therefore made up of three components: the costs of local computations, a latency term and a bandwidth term. The runtime of an algorithm is the sum of the runtimes of all its supersteps.

The *logP* model of Culler et al. [19] 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 which no other operation can be carried out. Once a message has been sent, the upper bound on latency L states how long a message takes at most to arrive at its destination. The destination processor

then has to receive the message. Hence, sending a message incurs a cost of at most $o+L+o$. 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.

The recently introduced *MapReduce* model by Dean and Ghemawat [20] 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. There are several ways to define the complexity of an algorithm in the MapReduce model [21]. 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 $\mathcal{O}\left(N^{1-\varepsilon}\right)$ for a problem of size N.

All models, (BSP, LogP and MapReduce) have in common that they ignore the topology of the communication network as well as the routing algorithm used to transmit the messages.

The communication model used in this paper and employed to estimate the runtime of the communication algorithms is described in detail in Section 4. The model draws from all of the presented models as well as models commonly used to analyze standard message-passing operations [22, 23], is very simple and still captures the aspects which are most important to us. We assume that the communication is round-based and that processors can communicate point-to-point by sending messages. In each round, one processor can either send or receive one message. The time to send a message of size m consists of a latency term and a bandwidth term and is $L+\frac{m}{B}$. The time needed for one round is the time taken to send the largest message of that round. As we focus on communication schemes, local computations are neglected. Nevertheless, we provide a comparison with actual runtimes at the end of Section 8.3.

2.2. Sparse Grids and the Combination Technique

The term sparse grids was coined by Zenger [3] for the solution of partial differential equations using a hierarchical function space decomposition. Some of the underlying ideas even date back to Smolyak [24]. Shortly thereafter, the so-called sparse grid combination technique was proposed by Griebel, Schneider and Zenger [4]. It assembles or approximates the sparse grid solution as a linear combination of many coarse, anisotropic full grids, called component grids [5]. 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 [25]. The sparse grid combination technique can be refined, too, but in a slightly more restricted, dimensionally adaptive way by adding complete anisotropic grids [5]. Furthermore, the combination coefficients of the combination technique can be chosen in a problem-dependent, optimal manner [26, 27]. Sparse grids and the sparse grid combination technique have been used to solve a variety of high-dimensional numerical problems including PDEs from mechanics, physics and financial mathematics [9, 28, 29, 30], real-time visualization [31, 32], machine learning problems [33], data mining [34, 35], etc.

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 [36, 37, 9, 38, 39, 40, 41] 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 [42] into the combination technique [43, 44, 45]. 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 and hence the need for efficient global communication schemes to do so. For time-dependent problems, it has been recognized that assembling the sparse grid solution is necessary after few time steps [46, 9]. Also, when the different component grids are distributed over a network of compute nodes, load balancing becomes an issue. The runtime of solving the 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 [41].

Early communication schemes used a farm of slaves to compute the component grid solutions and then assembled the sparse grid on a master [36, 37, 9]. 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 $\mathcal{O}\left(\sqrt{p}\right)$ by splitting the communication into 1-dimensional communication tasks was also proposed but not implemented [46]. Also, all-to-all communication

has been used to assemble the sparse grid solution on all component grids [47]. Recently, communication schemes that take advantage of the hierarchical structure of sparse grids were introduced [10]. This work improves these communication schemes and examines them in detail.

For the communication schemes to exploit the hierarchical structure of sparse grids, it is necessary that the component grids solutions are represented in the hierarchical basis. Hence, hierarchization and dehierarchization, the transformation from the full grid basis into the hierarchical basis and vice versa, are essential pre- and postprocessing steps for the presented communication schemes. Many implementations of the hierarchization and dehierarchization algorithms exist [33, 48, 49, 32, 50, 51], including versions that are tuned for GPUs [48, 32]. All these implementations take advantage of the so-called unidirectional principle which decomposes d-dimensional operators into multiple applications of 1-dimensional operators. Recently, the unidirectional hierarchization algorithm has been implemented for component grids such that it is within a factor of 1.5x of the lower bound imposed by the unidirectional principle [51]. Furthermore, for component grids, a hierarchization algorithm avoiding the unidirectional principle globally has been designed [52]. This algorithm reduces the leading term of the number of cache misses by a factor of d compared to the unidirectional lower bound.

3. Sparse Grids and Notation

As sparse grids [3, 24] and the combination technique [4] are not the main focus of this work, we refer to the survey [53] for details. The derivation of optimal communication algorithms does, however, require us to represent functions both with respect to anisotropic full grid spaces $V_{\boldsymbol{\ell}}$ and hierarchical increment spaces $W_{\boldsymbol{\ell}}$. Thus, we briefly introduce the required notation and sketch some important ideas. For simplicity, we only discuss sparse grids without boundary points. In Section 5.7.1 we modify the communication schemes such that they also work when boundary points are included. We also restrict the discussion to piecewise d-linear basis functions, but the observations can be extended to various kinds of standard basis functions.

The representation of a high-dimensional function can be based on a sparse grid discretization either in the hierarchical sparse grid basis or via a linear combination of partial solutions based on anisotropic full grids called component grids. We use the combination technique approach to decompose the whole problem into several problems on the component grids, which we compute independently. Every few time-steps, we synchronize those solutions in a reduce/broadcast step such that each grid point of each component grid reflects the current value of the global function. This global function is a linear combination of the functions represented by the component grids, see Figure 1 once more. One possibility to realize this reduce/broadcast step is to evaluate (interpolate) the function of each component grid at all sparse grid points and to collect those values to a vector. Then all these vectors are weighted and added componentwise. From the resulting sum we extract, for each component grid, the values at the points of the corresponding component grid. In this scheme many non-zero values are communicated, for each component grid many more than the number of its grid points. This can be avoided if the problem formulation is considered in the hierarchical basis. In this representation, values corresponding to sparse grid points missing from the component grid are zero, and need not to be communicated.

To this end, consider a conventional discretization of the d-dimensional space $\Omega := [0,1]^d$ on an anisotropic grid $C_{\boldsymbol{\ell}}$ with mesh-width $h_{\ell_k} := 2^{-\ell_k}$ and discretization level ℓ_k in dimension $k \in \{1,\dots,d\}$. A conventional anisotropic tensor product space $V_{\boldsymbol{\ell}}$ is then spanned by d-dimensional local tensor product basis functions $\varphi_{\boldsymbol{\ell},\mathbf{i}}(\mathbf{x}) := \prod_{k=1}^{d} \varphi_{\ell_k,i_k}(x_k)$ associated to the corresponding grid points $x_{\boldsymbol{\ell},\mathbf{i}}$, $x_{\ell_k,i_k} = i_k h_{\ell_k}$. For simplicity, think of piecewise d-linear functions and the classical full grid nodal basis as sketched in the left part of Figure 2. The same holds for other function spaces such as piecewise polynomial ones.

Alternatively, we can represent $V_{\boldsymbol{\ell}}$ by a unique decomposition into hierarchical increment spaces $W_{\boldsymbol{\ell}'}$ as $V_{\boldsymbol{\ell}} = \oplus_{\boldsymbol{\ell}' \le \boldsymbol{\ell}} W_{\boldsymbol{\ell}'}$. (Relations and functions on vectors are used component-wise, e.g. $\boldsymbol{\ell} \le \boldsymbol{\ell}' \Leftrightarrow \ell_i \le \ell'_i \; \forall i$ and $(\min\{\boldsymbol{\ell},\mathbf{k}\})_i = \min\{\ell_i,k_i\}$.) This formulation can be used as definition for the hierarchical increment spaces. The $W_{\boldsymbol{\ell}}$ are spanned by hierarchical basis functions, and we denote the coefficients for a certain $w_{\boldsymbol{\ell}} \in W_{\boldsymbol{\ell}}$ as (hierarchical) surpluses. Thus, we can represent each $f_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}}$ uniquely as $f_{\boldsymbol{\ell}}(\mathbf{x}) = \sum_{\boldsymbol{\ell}' \le \boldsymbol{\ell}} w_{\boldsymbol{\ell}'}(\mathbf{x})$ with $w_{\boldsymbol{\ell}'} \in W_{\boldsymbol{\ell}'}$. Figure 2 (right) shows grids and hierarchical piecewise linear basis functions corresponding to the hierarchical decomposition.

We would be interested in the full grid solution $u_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}}$ with $\ell_1 = \dots = \ell_d = n$ for some uniform discretization level n, which we cannot afford. But for sufficiently smooth functions, the sparse grid function $f_n \in V_n^{\mathrm{SG}}$, $V_n^{\mathrm{SG}} :=$

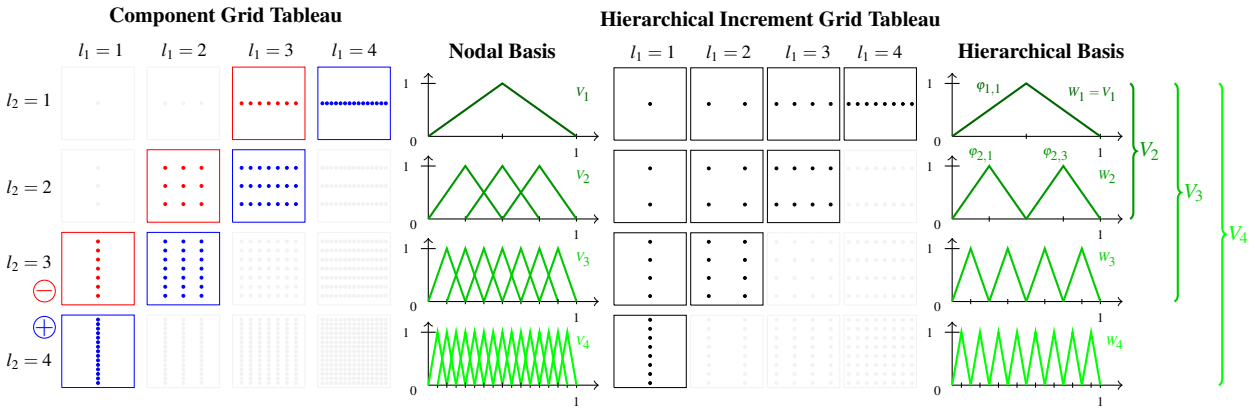

Figure 2: For different refinement level: Component grids (2D) with the respective 1-dimensional piecewise linear full grid nodal basis functions (left) and hierarchical increment grids with the respective hierarchical basis functions (right).

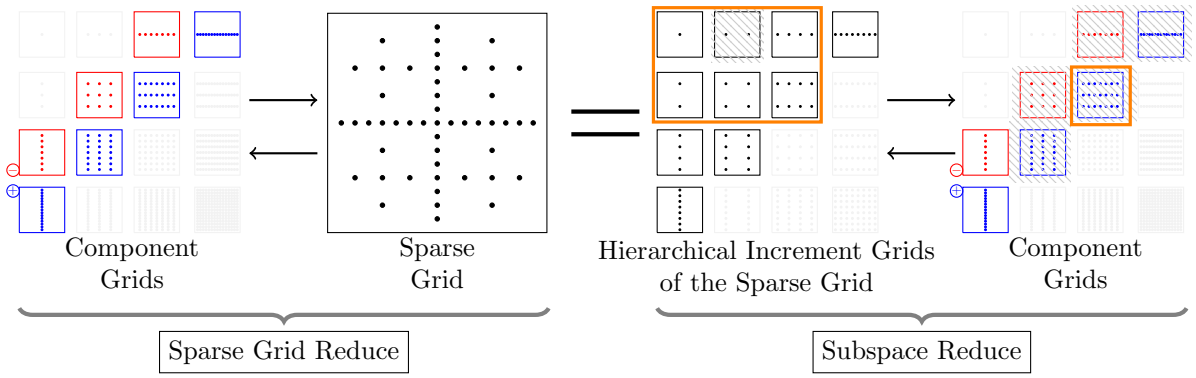

Figure 3: *Sparse Grid Reduce* and *Subspace Reduce*. The highlighted (orange) component grid can be uniquely decomposed into the 6 marked hierarchical increment grids. The hatched increment grid is part of the hatched component grids. The sparse grid solution can be assembled by a linear combination of the component grid solutions.

$\oplus_{\|\boldsymbol{\ell}\|_1 \leq n+d-1} W_{\boldsymbol{\ell}}$, yields an accurate approximation with orders of magnitudes less degrees of freedom by a careful a priori selection of hierarchical increment spaces.

The combination technique is based on full, but anisotropic and coarse grids $C_{\boldsymbol{\ell}}$, called *component grids*, for $\boldsymbol{\ell} \in \{\boldsymbol{\ell}' \in \mathbb{N}^d : \ n \leq \left\|\boldsymbol{\ell}'\right\|_1 \leq n+d-1\}$. The combination technique solution $f_n^{\mathrm{CT}} \in V_n^{\mathrm{SG}}$ can be obtained by a linear combination of the solutions on the component grids,

$$f_n^{\mathrm{CT}} = \sum_{j=0}^{d-1} (-1)^j \binom{d-1}{j} \sum_{\|\boldsymbol{\ell}\|_1 = n+d-1-j} f_{\boldsymbol{\ell}}\,, \qquad f_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}}\,. \tag{1}$$

The union of all corresponding grids $C_{\boldsymbol{\ell}}$ employed in this combination results in the respective sparse grid, see the example in Figure 3. For the communication task discussed in this paper, we can assume that the coefficients $(-1)^j \binom{d-1}{j}$ are already multiplied to the respective functions $f_{\boldsymbol{\ell}}$ and that thus the combination task is a summation.

We finally formulate two observations that we require later on for the derivation and analysis of the communication schemes. It is essential for both observations that we are working in the hierarchical and not in the nodal full grid basis.

Observation 1. *Let $f_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}} \subset V_n^{SG}$ be represented in the hierarchical sparse grid space $V_n^{SG} = \bigoplus_{\left\|\boldsymbol{\ell}'\right\|_1 \leq n+d-1} W_{\boldsymbol{\ell}'}$ as $f_{\boldsymbol{\ell}}(\mathbf{x}) = \sum_{\left\|\boldsymbol{\ell}'\right\|_1 \leq n+d-1} w_{\boldsymbol{\ell}'}(\mathbf{x})$ with $w_{\boldsymbol{\ell}'} \in W_{\boldsymbol{\ell}'}$. For all $\boldsymbol{\ell}' \not\leq \boldsymbol{\ell}$ it holds that $w_{\boldsymbol{\ell}'}(\mathbf{x}) \equiv 0$, i. e., all coefficients of $f_{\boldsymbol{\ell}}$ relating to the basis functions spanning $\bigoplus_{\boldsymbol{\ell}' \not\leq \boldsymbol{\ell}} W_{\boldsymbol{\ell}'}$ are 0.*

For a $V_{\boldsymbol{\ell}}$ and a function f define the projection $f|_{V_{\boldsymbol{\ell}}}$ by $f|_{V_{\boldsymbol{\ell}}} \in V_{\boldsymbol{\ell}}$ and $f|_{V_{\boldsymbol{\ell}}}(\mathbf{x}) = f(\mathbf{x}) \ \ \forall \mathbf{x} \in C_{\boldsymbol{\ell}}$.

Observation 2. *For $f = \sum_{\boldsymbol{\ell}' \le \boldsymbol{\ell}} w_{\boldsymbol{\ell}'} \in V_{\boldsymbol{\ell}}$ and a $V_{\mathbf{k}}$ it holds that $f|_{V_{\mathbf{k}}} = \sum_{\boldsymbol{\ell}' \le \mathbf{k} \wedge \boldsymbol{\ell}' \le \boldsymbol{\ell}} w_{\boldsymbol{\ell}'}$.*

By Observation 1, representing a function $f_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}}$ (which is given in the hierarchical basis) in the sparse grid space V_n^{SG} is achieved by filling in zeros.

Projecting $f_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}}$ from one anisotropic space to another one, $V_{\mathbf{k}}$, is done by sampling the hierarchical surpluses of $f_{\boldsymbol{\ell}}$ at the grid points common to $V_{\boldsymbol{\ell}}$ and $V_{\mathbf{k}}$ according to Observation 2. All grid points that are in $V_{\mathbf{k}}$ but not in $V_{\boldsymbol{\ell}}$ again have a 0 coefficient. In the nodal grid basis, both operations, i. e., representing $f_{\boldsymbol{\ell}}$ in the sparse grid space V_n^{SG} and projecting $f_{\boldsymbol{\ell}}$ onto $V_{\mathbf{k}}$, are more complicated and require interpolation.

Any function $f_{\boldsymbol{\ell}} \in V_{\boldsymbol{\ell}}$ is uniquely represented by its full grid or hierarchical coefficients. All we require for communication are the vectors of their coefficients. We assume that we can hierarchize and dehierarchize (transform from the full grid basis to the hierarchical basis and vice versa) for free, as node-local computations do not require communication. To describe and analyze the communication for the combination technique, we therefore define the notion of grids. 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. We summarize and define:

$C_{\boldsymbol{\ell}}$: the (component) grid corresponding to the anisotropic full grid space $V_{\boldsymbol{\ell}}$,
$H_{\boldsymbol{\ell}}$: the grid corresponding to the hierarchical increment space $W_{\boldsymbol{\ell}}$,
SG_n^d : the grid corresponding to the sparse grid space V_n^{SG} in d dimensions,
$\mathscr{C}_n^d$: the set of all component grids in Equation (1) for a sparse grid of level n in d dimensions, $\mathscr{C}_n^d := \{C_{\boldsymbol{\ell}} : n \le \|\boldsymbol{\ell}\|_1 \le n+d-1\}$, and
$\mathscr{H}_n^d$: the set of all hierarchical grids for V_n^{SG} in d dimensions, $\mathscr{H}_n^d := \{H_{\boldsymbol{\ell}} : \|\boldsymbol{\ell}\|_1 \le n+d-1\}$.

$|C_{\boldsymbol{\ell}}|$ denotes the number of grid points while $\left|\mathscr{C}_n^d\right|$ counts the number of component grids. In the analysis of the communication schemes and lower bounds, the set of component grids $\mathbf{CG}(H_{\boldsymbol{\ell}})$ that contributes to a particular hierarchical increment space $H_{\boldsymbol{\ell}} \in \mathscr{H}_n^d$, i. e., that contain the grid points of $H_{\boldsymbol{\ell}}$, is important:

$$\mathbf{CG}(H_{\boldsymbol{\ell}}) = \{C_{\boldsymbol{\ell}'} \in \mathscr{C}_n^d : H_{\boldsymbol{\ell}} \subseteq C_{\boldsymbol{\ell}'}\} = \{C_{\boldsymbol{\ell}'} \in \mathscr{C}_n^d : \boldsymbol{\ell}' \ge \boldsymbol{\ell}\} \ .$$

For a set $A \subset \mathscr{C}_n^d$ of component grids we furthermore define the set of grid points shared with the component grids in $\mathscr{C}_n^d \setminus A$:

$$\mathbf{sharedG}(A) = \bigcup_{C_{\boldsymbol{\ell}} \in A} C_{\boldsymbol{\ell}} \ \cap \bigcup_{C_{\boldsymbol{\ell}} \in \left(\mathscr{C}_n^d \setminus A\right)} C_{\boldsymbol{\ell}} \ = \bigcup_{\substack{H_{\boldsymbol{\ell}} \in \mathscr{H}_n^d : \\ \mathbf{CG}(H_{\boldsymbol{\ell}}) \cap A \neq \emptyset,\ \mathbf{CG}(H_{\boldsymbol{\ell}}) \not\subseteq A}} H_{\boldsymbol{\ell}} \ .$$

If the component grids of A are stored on one set of nodes while all other component grids $\mathscr{C}_n^d \setminus A$ 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.

4. Model, Task and Cost Measures

We work with a standard simple communication model similar to the BSP and the logP model. It is equivalent to the models typically used to analyze message passing operations [22, 23]. The model is designed to capture the trade-off between sending many small messages versus sending few big messages. The details and assumptions of the model are: the communication nodes (in the following only nodes) are uniformly connected in a fully connected network topology, 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 exactly one node per component grid is responsible for the communication. This work focuses on basic communication patterns and does neither address load balancing nor the cases that several component grids are computed on one node or that a component grid is distributed onto several nodes. As we only study communication patterns, the computations on a node, e. g., summation, copying, or pre- and postprocessing steps like hierarchization and dehierarchization, are not considered.

We study the following communication task:

Input: A set of nodes $N_{\boldsymbol{\ell}}$ ($\boldsymbol{\ell}$ such that $C_{\boldsymbol{\ell}} \in \mathscr{C}_n^d$). Each node $N_{\boldsymbol{\ell}}$ stores the values of $f_{\boldsymbol{\ell}}$ (e.g. the solution of a PDE) at the grid points of component grid $C_{\boldsymbol{\ell}}$.

Output: For all $N_{\boldsymbol{\ell}}$: Node $N_{\boldsymbol{\ell}}$ stores $f_n^{\mathrm{CT}}\big|_{V_{\boldsymbol{\ell}}}$, i.e. the values of the combination technique solution f_n^{CT} (see (1)) at the grid points of component grid $C_{\boldsymbol{\ell}}$.

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,

$$L \cdot (\text{number of rounds}) + \frac{1}{B} \cdot (\text{makespan volume}) \ . \tag{2}$$

5. Algorithms and Lower Bounds

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. For the analysis carried out in this paper, we assume that the communication task is split into two distinct phases, reduce and broadcast. The reduce step assembles the values of the combination technique solution f_n^{CT}, as in (1), for all grid points $\mathbf{x} \in \mathrm{SG}_n^d$. For *Subspace Reduce* the combination solution is not necessarily gathered on exactly one node, but is distributed over several nodes. The broadcast phases distributes the combination technique solution f_n^{CT} to all compute nodes. Splitting the communication task into two distinct phases allows 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.

The only communication operations that the communication schemes perform are *AllReduce*-operations. Thus, their implementation has a significant impact on the overall performance of our algorithms. We will discuss different implementations of *AllReduce* and their effects on the presented communication schemes in Section 5.1.

The two presented communication schemes are depicted in Figure 3. *Sparse Grid Reduce* expands each component grid to the whole sparse grid and then on all compute nodes performs a single *AllReduce* on the whole sparse grid. *Sparse Grid Reduce* minimizes the number of messages sent and the number of rounds at the expense of a larger communication volume.

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 reduce the values of this increment space using *AllReduce*. For *Subspace Reduce* the values of the component grids have to be hierarchized, i.e. they have to represented in the

Table 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. For the upper bounds we assume that an *AllReduce*-operation is split into a reduce and a broadcast step, each implemented using binomial trees.

Arbitrary Dimension	Lower Bound	Sparse Grid Reduce	Subspace Reduce	Parallel Subspace Reduce
Messages sent:	$2 \cdot \left(\left\|\mathscr{C}_n^d\right\| - 1\right)$	$2 \cdot \left(\left\|\mathscr{C}_n^d\right\| - 1\right)$	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} (\mathbf{CG}(H) - 1)$	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} (\mathbf{CG}(H) - 1)$
Number of rounds:	$2 \cdot \log_2 \left\|\mathscr{C}_n^d\right\|$	$2 \cdot \log_2 \left\|\mathscr{C}_n^d\right\|$	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \log_2 (\mathbf{CG}(H))$	$2 \cdot \sum_{\ell=1}^{n-1} \max_{H \in \mathscr{H}_\ell^d} (n-\ell)^{d-1} \cdot \log_2(\mathbf{CG}(H))$
Total volume:	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \|H\| (\mathbf{CG}(H) - 1)$	$2 \cdot \left(\left(\left\|\mathscr{C}_n^d\right\| - 1\right) \left\|\mathrm{SG}_{n-1}^d\right\|\right)$	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \|H\| \cdot (\mathbf{CG}(H) - 1)$	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \|H\| \cdot (\mathbf{CG}(H) - 1)$
Makespan volume:	$2 \cdot \max_{C \subset \mathscr{C}_n^d} \mathbf{sharedG}(C)$	$2 \cdot \left\|\mathrm{SG}_{n-1}^d\right\| \cdot \log_2 \left(\left\|\mathscr{C}_n^d\right\|\right)$	$2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \|H\| \cdot \log_2 (\mathbf{CG}(H))$	$2 \cdot \sum_{\ell=1}^{n-1} \max_{H \in \mathscr{H}_\ell^d} \|H\| \cdot (n-\ell)^{d-1} \cdot \log_2(\mathbf{CG}(H))$
Maximum node size:	$\max_{C \in \mathscr{C}_n^d} \|C\|$	$\left\|\mathrm{SG}_{n-1}^d\right\| + \max_{C \in \mathscr{C}_n^d} \|C\|$	$\max_{C \in \mathscr{C}_n^d} \|C\| + \max_{H \in \mathscr{H}_{n-1}^d} \|H\|$	$\max_{C \in \mathscr{C}_n^d} \|C\| + \max_{H \in \mathscr{H}_{n-1}^d} \|H\|$
Communication nodes:	$\left\|\mathscr{C}_n^d\right\|$	$\left\|\mathscr{C}_n^d\right\|$	$\left\|\mathscr{C}_n^d\right\|$	$\left\|\mathscr{C}_n^d\right\|$

hierarchical basis instead of the full grid basis. *Subspace Reduce* minimizes the total communication volume while the number of messages and rounds increases. We also present *Parallel Subspace Reduce*, a parallel version of *Subspace Reduce*. It additionally achieves an asymptotically optimal makespan volume.

Table 1 presents the lower bounds and the complexities of the approaches for the different cost measures, as derived later in this section and in Section 6.

We generalize the approaches to sparse grids with boundary points and to sparse grids that have a minimum level. Generalizations to other types of sparse grids are straightforward as long as the adaptivity always includes whole increment spaces.

5.1. Analysis and Discussion of AllReduce

The building block of our algorithms is *AllReduce* that solves the following task: assume m communication nodes want to reduce a vector $\mathbf{v}$. Initially, each node i holds its version $\mathbf{v}_i$ of the vector. After the execution of *AllReduce* each node has the sum $\mathbf{v}_{\text{res}} := \sum_{i=1}^{m} \mathbf{v}_i$.

We split this task into a reduce phase that creates $\mathbf{v}_{\text{res}}$, 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 and the total communication volume by binomial trees in the following way: The communication nodes are arranged in a binomial tree of height $h = \lceil \log_2 m \rceil$. During each round of the reduce phase, every leaf sends its vector $\mathbf{v}_i$ of partial sums to its parent node, who adds it to its own vector. For $m < 2^h$ the binomial tree may be incomplete. In that case, one leaf delays its message until the next round. After $\lceil \log_2 m \rceil$ rounds, the root node has $\mathbf{v}_{\text{res}}$. The makespan volume of the reduce phase is $|\mathbf{v}| \cdot \lceil \log_2 m \rceil$.

In the broadcast phase the messages are sent in reverse order, copying the combined vector $\mathbf{v}_{\text{res}}$ to all nodes in $\lceil \log_2 m \rceil$ rounds. For both phases together the number of rounds is $2 \cdot \lceil \log_2 m \rceil$ and the makespan volume is $2 \cdot |\mathbf{v}| \cdot \lceil \log_2 m \rceil$. Note that the reduce and broadcast phase are time inverse to each other.

In order to see that binomial trees optimally solve both, the reduce and the broadcast step, note that: at least $m-1$ messages, one less than the number of participating nodes, need to be sent in each of the two steps. Also, each entry of $\mathbf{v}$ must be sent at least $(m-1)$ times. Hence, the total volume for each phase is $|\mathbf{v}| \cdot (m-1)$. As the number of nodes which have yet provided 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 paper we assume that *AllReduce* is split into these two distinct phases, reduce and broadcast. Furthermore, we assume, that *AllReduce* is implemented using binomial trees. In fact many common implementations of *AllReduce* use binomial trees. However, in practice, also implementations of *AllReduce* are employed that merge both phases and do not use binomial trees. Furthermore, there are implementations that switch between different algorithms depending on the size of $\mathbf{v}$ and the number of communication partners. Taking this into account for our analysis, the complexity of the model would increase without bringing any further insights. The model [22, 23] used

to analyze message-passing operations like *AllReduce* assumes that communication links are bidirectional, i.e. a node can send and receive a message simultaneously. Hence, this model differs from the model of Section 4.

Given bidirectional communication links, *AllReduce* can merge the reduce and the broadcast phase by means of a recursive-doubling approach [22]. 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 $|\mathbf{v}| \cdot \lceil \log_2 m \rceil$. As the lower bound of $\log_2 m$ rounds for one phase, either reduce or broadcast, carries over to the *AllReduce*-task, an implementation of *AllReduce* based on a recursive doubling approach works in the minimum number of rounds. However, the total communication volume and the total number of messages increases, because 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 [22]. Both operations can be implemented by passing chunks of size $|\mathbf{v}| \big/ m$ between the processors in a round-robin fashion in $m-1$ rounds. Such an implementation of *AllReduce* communicates the minimum total volume [23]. The optimal makespan volume comes at the price of an increased number of rounds and messages sent.[1]

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.

5.2. 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, $\mathscr{C}$)*: perform an *AllReduce* operation for the set S using the communication nodes $\mathscr{C}$. In more detail: if S_j is the copy of S stored initially on $C_j \in \mathscr{C}$ 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 $s_{\text{res},i} = \sum_{j, C_j \in \mathscr{C}} s_{j,i}$ and the resulting set $S_{\text{res}} = \bigcup_{i \in I} s_{\text{res},i}$ is stored at all nodes $C_j \in \mathscr{C}$.
- *Extract S from Buffer*: Copy the values of the set S from the buffer to the corresponding positions of the local copy of S.

5.3. Lower Bounds

The lower bounds for the different cost measures are summarized in Table 1. For a sparse grid of level n and dimension d the communication task consists of $\left|\mathscr{C}_n^d\right|$ component grids and hence this is the minimum number of communication nodes required. Also, the main memory of the communication nodes has to be large enough to store at least the largest component grid, $\max_{C_{\boldsymbol{\ell}} \in \mathscr{C}_n^d} |C_{\boldsymbol{\ell}}|$. 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 $H_{\mathbf{1}}$ is part of all $C_{\boldsymbol{\ell}} \in \mathscr{C}_n^d$ we need at least $\left|\mathscr{C}_n^d\right| - 1$ messages to calculate the combination technique solution at the grid point of this increment space. Furthermore, since the number of nodes over which the overall sum for $H_{\mathbf{1}}$ is currently still split can at best be halved in each round, there are at least $\left\lceil \log_2 \left|\mathscr{C}_n^d\right| \right\rceil$ communication rounds. To give a lower bound of the

[1] The reduce-scatter and the allgather operation can also be implemented by recursive halving or recursive doubling [22] but because of the two distinct phases the number of rounds is still twice that of the lower bound.

Algorithm 1: *Sparse Grid Reduce* for component grid $C_{\boldsymbol{\ell}}$.

foreach $H_{\mathbf{k}} \subset SG^d_{n-1}$ **do**
  if $H_{\mathbf{k}} \subseteq C_{\boldsymbol{\ell}}$ **then**
    Buffer $H_{\mathbf{k}} \subseteq C_{\boldsymbol{\ell}}$
  else
    Buffer $H_{\mathbf{k}} \equiv 0$
AllReduce (SG^d_{n-1}, $\mathscr{C}^d_n$)
foreach $H_{\mathbf{k}} \subseteq C_{\boldsymbol{\ell}}$ **do**
  Extract $H_{\mathbf{k}}$ *from Buffer*

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,

$$\sum_{H_{\boldsymbol{\ell}} \in \mathscr{H}^d_n} |H_{\boldsymbol{\ell}}| \cdot (\mathbf{CG}(H_{\boldsymbol{\ell}}) - 1) = \left(\sum_{C_{\boldsymbol{\ell}} \in \mathscr{C}^d_n} |C_{\boldsymbol{\ell}}| \right) - \left| \mathrm{SG}^d_n \right| . \tag{3}$$

A trivial lower bound for the makespan communication volume is $\max_{C_{\boldsymbol{\ell}} \subset \mathscr{C}^d_n} |\mathbf{sharedG}(C_{\boldsymbol{\ell}})|$.

5.4. Sparse Grid Reduce (Algorithm 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 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 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 SG^d_{n-1} which are not in the component grid of the respective node.

5.5. Subspace Reduce (Algorithm 2) – Communicating Each Hierarchical Increment Space Separately

Subspace Reduce is the other extreme. Here, we split the communication task into subtasks determined by the hierarchical increment spaces as suggested by Observation 2. First, fix an order of the hierarchical increment spaces. Then, for all hierarchical increment spaces $H_{\boldsymbol{\ell}} \in \mathscr{H}^d_{n-1}$ perform an *AllReduce* employing all component grids containing $H_{\boldsymbol{\ell}}$, i.e. the nodes $\mathbf{CG}(H_{\boldsymbol{\ell}})$. 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 2.

The complexities of *Subspace Reduce* with respect to the different cost measures are summarized in Table 1. As only nodes that require a particular $H_{\boldsymbol{\ell}}$ 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) *Subspace Reduce* we make the pessimistic assumption that all increment spaces are reduced in serial. In fact,

Algorithm 2: *SubSpace Reduce* for component grid $C_{\boldsymbol{\ell}}$.

```
foreach H_k ∈ H^d_{n-1} do
    if H_k ⊆ C_ℓ then
        Buffer H_k
        AllReduce(H_k, CG(H_k))
        Extract H_k from Buffer
```

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 there is a jump in the lexicographic order), and hence *Subspace Reduce* has to reduce the increment spaces almost in serial. Section 5.6 introduces a simple order of the hierarchical increment spaces that allows for the parallel execution of the *AllReduce*-operations.

Subspace Reduce relies on working on the hierarchical surpluses. By Observation 1, the hierarchical surplus of a grid point not in a certain component grid $C_{\boldsymbol{\ell}}$ is 0 for this component grid. This allows us to limit the reduce procedure for an increment space $H_{\boldsymbol{\ell}}$ to the subset $\mathbf{CG}(H_{\boldsymbol{\ell}})$ 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 *Subspace Reduce*.

5.6. Enabling Parallelism: the Order of Increment Spaces for Parallel Subspace Reduce

Subspace Reduce (Algorithm 2) does not specify the order in which the increment spaces are reduced and the analysis of the original *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 *AllReduce*-operations.

Denote by $a \in \{0,\dots,n-1\}$ the distance from the sparse grid level n and look at 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 $\boldsymbol{\ell}$ of all increment spaces of fixed level sum into a $d-1$ dimensional space by projecting along dimension d, i.e. by ignoring the dth 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 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 $\boldsymbol{\ell}_{(d-1)}$ and the original level vector $\boldsymbol{\ell}$. 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_i the unit vector in dimension i. Any pair of increment spaces of level sum $(n+d-1)-a$ whose level vector is at stride $(a+1)\cdot e_i$ for some $i \in \{1,\dots,d-1\}$ is not contained in the same component grid and can be reduced in parallel (see also Figure 4). 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 3. *Fix a distance $a \in \{0,\dots,n-1\}$ from the maximum refinement level $n+d-1$ and one level vector $\boldsymbol{\ell}$ at that distance a, i.e. $\|\boldsymbol{\ell}\|_1 = (n+d-1)-a$. Out of all increment spaces $H_{\boldsymbol{\ell}'}$ with $\|\boldsymbol{\ell}'\|_1 = (n+d-1)-a$, those whose projected level vector $\boldsymbol{\ell}'_{(d-1)}$ suffices*

$$\boldsymbol{\ell}'_{(d-1)} = \boldsymbol{\ell}_{(d-1)} + (a+1)\cdot\sum_{i=1}^{d-1} k_i \cdot e_i \qquad \text{for} \quad k_i \in \mathbb{Z} \tag{4}$$

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.

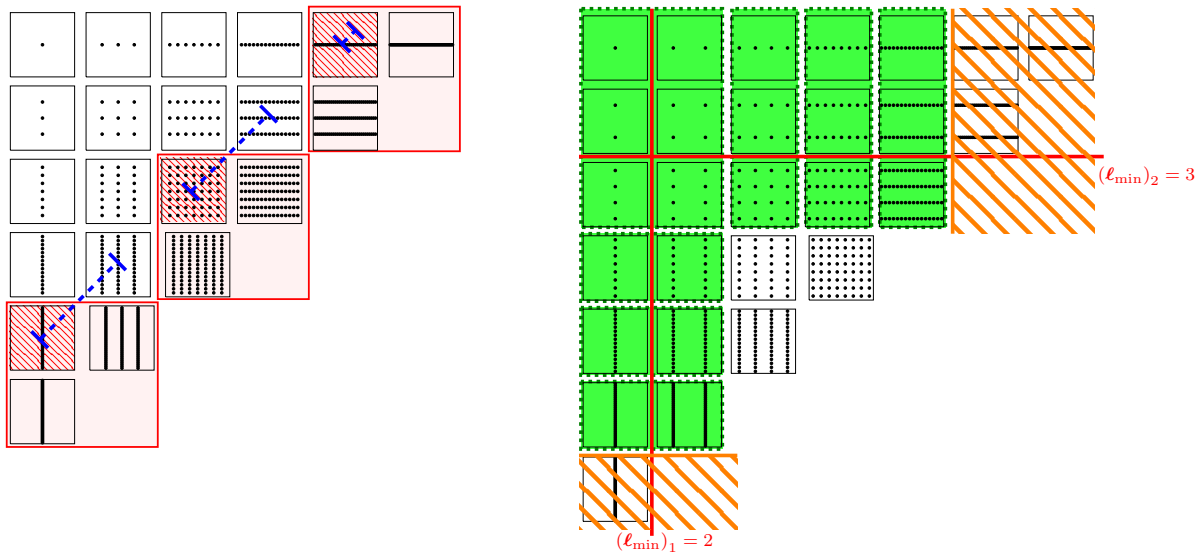

Figure 4: *Left:* reducing increment spaces in parallel for $a = 1$. The increment spaces corresponding to the shaded component grids can be reduced in parallel requiring the framed component grids. In 2 dimensions the $d-1$ dimension hypercubes correspond to intervals (dashed). Increment spaces within the same interval cannot be reduced in parallel as they share component grids. *Right:* adjusting increment spaces when introducing a minimum level (here $\boldsymbol{\ell}_{\min} = (2,3)$). 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 $\boldsymbol{\ell}$ suffices $\boldsymbol{\ell} \geq \boldsymbol{\ell}_{\min} + \mathbf{1}$ remain unchanged.

To reduce increment spaces of level sum $n+d-2$, i.e. $a = 1$, we need the component grid $C_{\boldsymbol{\ell}}$ and also the component grids that are refined once further than $\boldsymbol{\ell}$ in every dimension. Hence, if we reduce $H_{\boldsymbol{\ell}}$, then $C_{\boldsymbol{\ell}}$ and all component grids $C_{\boldsymbol{\ell}'}$ with $\boldsymbol{\ell}' = \boldsymbol{\ell} + e_i$ for any $1 \leq i \leq d$ are participating in the current reduce. This means, all $H_{\boldsymbol{\ell}'}$ with $\boldsymbol{\ell}'_{(d-1)} = \boldsymbol{\ell}_{(d-1)} + 2\sum_{i=1}^{d-1} k_i \cdot e_i$ for $k_i \in \mathbb{Z}$ can be reduced in parallel.

To reduce increment spaces of level sum $n+d-3$, i.e. $a = 2$, we need $C_{\boldsymbol{\ell}}$ and also at most the component grids that are twice further refined than $\boldsymbol{\ell}$. Hence, if we reduce $H_{\boldsymbol{\ell}}$ then the component grids $C_{\boldsymbol{\ell}'}$ with $\boldsymbol{\ell}' = \boldsymbol{\ell} + \sum_{i=1}^{d} b_i \cdot e_i$ for $b_i \in \{0,1,2\}$ 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 $H_{\boldsymbol{\ell}'}$ for $\boldsymbol{\ell}'_{(d-1)} = \boldsymbol{\ell}_{(d-1)} + 3\sum_{i=1}^{d-1} k_i \cdot e_i$ for $k_i \in \mathbb{Z}$ in parallel.

For general a, we need at most the component grids that are a times further refined than $\boldsymbol{\ell}$. Hence, if we reduce $H_{\boldsymbol{\ell}}$ then the component grids $C_{\boldsymbol{\ell}'}$ with $\boldsymbol{\ell}' = \boldsymbol{\ell} + \sum_{i=1}^{d} b_i \cdot e_i$ for $b_i \in \{0,1,2,\ldots,a\}$ are a super set for the component grids needed for this reduce. Therefore we can reduce all $H_{\boldsymbol{\ell}'}$ for $\boldsymbol{\ell}'_{(d-1)} = \boldsymbol{\ell}_{(d-1)} + (a+1) \cdot \sum_{i=1}^{d-1} k_i \cdot e_i$ for $k_i \in \mathbb{Z}$ 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 3 does not state all parallelism that can be exploited for the *AllReduce*-operations as the given superset for the component grids necessary for a reduce step is slightly pessimistic.

The parallelism described by Observation 3 can be exploited by *Subspace Reduce* (Algorithm 2) when the hierarchical increment spaces $H_{\boldsymbol{\ell}}$ are ordered by:

- level sum $\|\boldsymbol{\ell}\|_1$ increasing from d to $n+d-1$ (distance a decreasing from $n-1$ to 0),
- relative position of $H_{\boldsymbol{\ell}}$ 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 $\|\boldsymbol{\ell}\|_1 = (n+d-1)-a$.

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 out of the scope of this work.

5.7. Adapting the Algorithms to Treat Generalizations of Sparse Grids

This section describes the necessary modifications such that the two algorithms *Sparse Grid Reduce* and *Subspace Reduce* can handle sparse grids with boundary, with minimum level or both.

5.7.1. Boundary Treatment

There are three main approaches to equip sparse grids with boundaries. 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 5 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 (boundary of level-1 type) this section describes the necessary modifications of the algorithms for all three types of boundary functions.

The first approach modifies the outermost basis functions of each level of the basic one-dimensional setting of the tensor product approach such that they extrapolate towards the boundary. For these boundary functions no modifications to the algorithms are necessary as no new grid points are introduced.

The second approach adds new boundary functions to existing hierarchical increment spaces and hence the structure of the communication task does not change. In detail, the new boundary basis functions are assigned to level 1. Hence, this kind of boundary is called level-1 type and does not change the set of increment spaces $\mathscr{H}_n^d$. The structure of the communication is not altered, and still the same nodes need to communicate. Only the increment spaces $H_{\boldsymbol{\ell}} \in \mathscr{H}_n^d$ themselves change. For the communication task, their increase in size is relevant as it changes the size of the messages that are passed. Whenever $\min(\boldsymbol{\ell}) = 1$, then the increment space $H_{\boldsymbol{\ell}}$ is affected by the boundary and has larger size. Similarly, the set of component grids $\mathscr{C}_n^d$ does not change while the component grids $C_{\boldsymbol{\ell}}$ now all include boundary vertices and hence have increased size. The size of the sparse grid SG_{n-1}^d also increases due to the boundary. As the set of hierarchical increment spaces is not altered for boundary of level-1 type, both algorithms, *Sparse Grid Reduce* and *Subspace Reduce*, work as before in the sense that still the same nodes need to communicate. Only the size of the messages changes.

In the third approach new hierarchical increment spaces are created for the boundary basis functions. The new boundary basis functions are added on a new level 0, and hence this boundary is also called of level-0 type. For both schemes, *Sparse Grid Reduce* and *Subspace Reduce*, this approach changes the structure of the communication scheme as the level-vector $\boldsymbol{\ell}$ of $H_{\boldsymbol{\ell}} \in \mathscr{H}_{n-1}^d$ and $C_{\boldsymbol{\ell}} \in \mathscr{C}_n^d$ can now also attain the value 0. This means that additional increment spaces as well as component grids with $\min(\boldsymbol{\ell}) = 0$ are added to the scheme. As a result, the size of SG_{n-1}^d as well as the size of all component grids changes as they all now include boundary points. The size of the former increment spaces $H_{\boldsymbol{\ell}} \in \mathscr{H}_{n-1}^d$ does not change, however. Only new increment spaces are added. Still, with the modified definitions of $\mathscr{H}_{n-1}^d$, $H_{\boldsymbol{\ell}}$, $\mathscr{C}_n^d$, $H_{\boldsymbol{\ell}}$ and SG_{n-1}^d, *Sparse Grid Reduce* and *Subspace Reduce* work as given in Algorithm 1 and Algorithm 2.

5.7.2. Introducing and Handling a Minimum Level

A minimum level $\boldsymbol{\ell}_{\min}$ is a threshold for the refinement level of the component grids. It can be used to exclude component grids from the combination that do not meet this minimum refinement criteria. Although in theory this will deteriorate the approximation quality of the combination solution, for practical applications it can be necessary to use a minimum refinement level in some dimensions to represent the underlying physical properties or to avoid stability problems of the numerical methods. For example, experiments [7] conducted with GENE [6] showed that

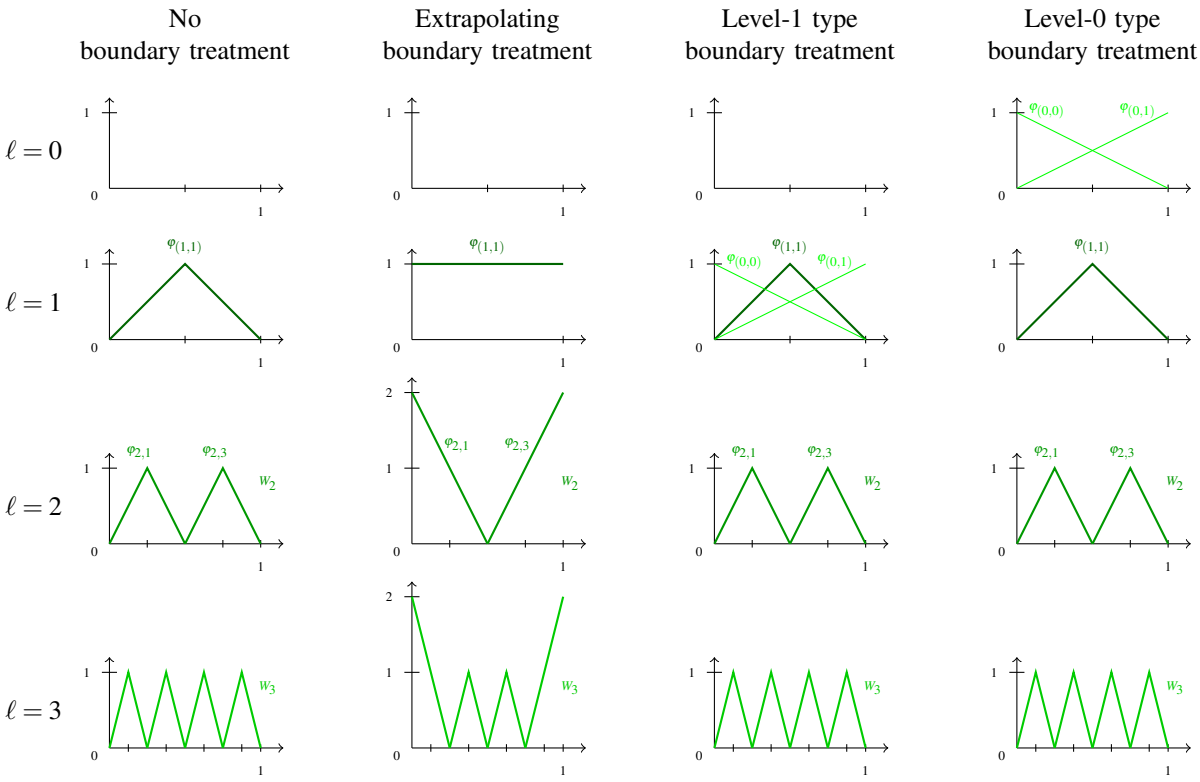

Figure 5: Different kinds of boundary treatment for the one-dimensional basis functions.

highly anisotropic grids should be excluded from the combination as a minimum discretization is required to represent physical properties.

For our experiments in Section 8 we additionally used the minimum level to restrict the number of component grids. Increasing the level n of the sparse grid increases the number of component grids $\left|\mathscr{C}_n^d\right|$ in the combination. Since we assume that each component grid is stored on its own shared memory node, we needed to restrict the number of grids to be able to run experiments with high discretization levels at all. Note that in real-world settings already two grids might exceed the memory of one node. To clarify the magnitude: for $n = 10$ there are 136 component grids for $d = 3$, 1 876 component grids for $d = 5$ and 92 378 component grids for $d = 10$ if we assume no boundary or boundaries of level-1 type. If the level sum of the minimum level $\|\boldsymbol{\ell}_{\min}\|_1$ 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. For example, for $d = 5$ there are 456 component grids for $n = 7$ and $\boldsymbol{\ell}_{\min} = (1,1,1,1,1)$ as well as for $n = 8$ and $\boldsymbol{\ell}_{\min} = (1,1,1,1,2)$ and $n = 17$ and $\boldsymbol{\ell}_{\min} = (3,3,3,3,3)$. Although increasing the minimum level when increasing n would be uncommon for practical application, here it helps us to obtain comparable experiments. Furthermore, increasing the minimum level by the same amount as the sparse grid level n has the effect that the communication task keeps the same structure: the same nodes need to exchange messages, only the size of the messages differs. We first introduce the minimum level for a sparse grid without boundary and then generalize to sparse grids with boundary points.

Formally, let $\boldsymbol{\ell}_{\min}$ denote the minimum level. Then, the altered set of component grids $\mathscr{C}^d_{(n,\boldsymbol{\ell}_{\min})}$ is

$$\mathscr{C}^d_{(n,\boldsymbol{\ell}_{\min})} = \left\{ C_{\boldsymbol{\ell}} \in \mathscr{C}^d_n : \boldsymbol{\ell} \geq \boldsymbol{\ell}_{\min} \right\} . \tag{5}$$

The component grids $C_{\boldsymbol{\ell}}$ themselves remain unchanged. However, for the hierarchical increment spaces, we face two issues. First, some increment spaces become no longer necessary, because the component grids that contained them are not in $\mathscr{C}^d_n$ any more. Second, to reduce the number of messages passed by *Subspace Reduce* we want to merge increment spaces $H_{\boldsymbol{\ell}}$ that belong to the same set of component grids $\mathbf{CG}(H_{\boldsymbol{\ell}})$ due to the new minimum level restriction.

Figure 4 (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 $\boldsymbol{\ell}$ define the set

$$I_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})} = \{i \in \{1,\dots,d\} : \ell_i = (\ell_{\min})_i\} \tag{6}$$

stating which dimensions are refined minimally. With that on hand we can create sets $K_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})}$ of level vectors whose corresponding increment spaces are going to be merged:

$$K_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})} = \left\{\boldsymbol{\ell}' : \left(1 \le \ell'_i \le \ell_i \text{ for } i \in I_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})}\right) \wedge \left(\ell'_j = \ell_j \text{ for } j \notin I\right)\right\} . \tag{7}$$

The sets $K_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})}$ can be used to assemble the merged increment spaces $H_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})}$ which merge the increment spaces $H_{\boldsymbol{\ell}'}$ with a level $\boldsymbol{\ell}'$ below the minimum level with the increment space $H_{\boldsymbol{\ell}}$ of smallest level $\boldsymbol{\ell}$ which is at the threshold $\ell_{\min}$ and larger than $\boldsymbol{\ell}'$,

$$H_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})} = \bigcup_{\boldsymbol{\ell}' \in K_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})}} H_{\boldsymbol{\ell}'} . \tag{8}$$

The hierarchical increment spaces $H_{\boldsymbol{\ell}}$ for which $I_{\boldsymbol{\ell}} \neq \varnothing$ 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 $\mathscr{H}^d_{(n,\boldsymbol{\ell}_{\min})}$ taking the minimum level $\boldsymbol{\ell}_{\min}$ into account is

$$\mathscr{H}^d_{(n,\boldsymbol{\ell}_{\min})} = \left\{H_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})} : \ell \text{ such that } H_{\boldsymbol{\ell}} \in \mathscr{H}^d_n \text{ and } \boldsymbol{\ell} \ge \boldsymbol{\ell}_{\min}\right\} . \tag{9}$$

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 $\boldsymbol{\ell}_{\min}$ is given by

$$\mathrm{SG}^d_{(n,\boldsymbol{\ell}_{\min})} = \bigcup_{H_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})} \in \mathscr{H}^d_{(n,\boldsymbol{\ell}_{\min})}} H_{(\boldsymbol{\ell},\boldsymbol{\ell}_{\min})} . \tag{10}$$

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 5.7.1 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 i we merge all 1-dimensional increment spaces in dimension i up to $(\boldsymbol{\ell}_{\min})_i$ to one increment space. The boundary basis functions of type 1 have also been merged with the root basis function on level 1.

6. Runtime Analysis

This section elaborates on the generic runtime formulas for *Sparse Grid Reduce* and *Subspace Reduce* given in Table 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. Similar formulas can be derived if boundary points or a minimum level is included. For the experimental part in Section 8, 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 4: 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 (2). All the formulas build on the assumption (Section 5.1) 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. Finally, we analyze *Parallel Subspace Reduce* in the communication model.

6.1. Precise Communication-Time Formulas for Sparse Grids Without Boundary and Without Minimum Level

Denote by A_k^d 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 $\mathscr{C}_k^d \setminus \mathscr{C}_{k-1}^d$ and $\mathscr{H}_k^d \setminus \mathscr{H}_{k-1}^d$ 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_k^d is given by

$$A_k^d = \binom{k+d-2}{d-1}. \tag{11}$$

The size of a sparse grid without boundary and minimum level is given by [53] as

$$\left|\mathrm{SG}_n^d\right| = \sum_{i=0}^{n-1} 2^i \cdot \binom{d-1+i}{d-1} = 2^n \cdot \left(\frac{n^{d-1}}{(d-1)!} + \mathscr{O}\left(n^{d-2}\right)\right). \tag{12}$$

As standard in the sparse grid literature, we assume that the dimension d is constant and hence omit this constant in the $\mathscr{O}$-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

$$\begin{aligned}
\mathit{SG\ Reduce}(n,d) &\overset{\text{Table 1}}{=} 2 \cdot L \cdot \left\lceil \log_2 \left|\mathscr{C}_n^d\right| \right\rceil + 2 \cdot \frac{8}{B} \cdot \left\lceil \log_2 \left|\mathscr{C}_n^d\right| \right\rceil \cdot \left|\mathrm{SG}_{n-1}^d\right| = \\
&= 2 \cdot \left\lceil \log_2 \left|\mathscr{C}_n^d\right| \right\rceil \left(L + \frac{8}{B} \cdot \left|\mathrm{SG}_{n-1}^d\right|\right) = \\
&= 2 \cdot \left\lceil \log_2 \left(\sum_{i=n-d+1}^{n} A_i^d\right) \right\rceil \cdot \left(L + \frac{8}{B} \cdot \left|\mathrm{SG}_{n-1}^d\right|\right) = \\
&= 2 \cdot \left\lceil \log_2 \left(\sum_{i=n-d+1}^{n} \binom{i+d-2}{d-1}\right) \right\rceil \cdot \left(L + \frac{8}{B} \cdot \sum_{i=0}^{n-2} 2^i \cdot \binom{d-1+i}{d-1}\right).
\end{aligned} \tag{13}$$

For $n < d$ the first sum has to start at $i = 1$ instead of $i = n-d+1$.

The runtime for *Subspace Reduce* is

$$\begin{aligned}
\mathit{Subspace\ Reduce}(n,d) &\overset{\text{Table 1}}{=} 2 \cdot L \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \lceil \log_2 |CG(H)| \rceil + 2 \cdot \frac{8}{B} \cdot \sum_{H \in \mathscr{H}_{n-1}^d} |H| \cdot \lceil \log_2 |CG(H)| \rceil = \\
&= 2 \cdot \sum_{H \in \mathscr{H}_{n-1}^d} \left[\left(L + \frac{8}{B} |H|\right) \cdot \lceil \log_2 |CG(H)| \rceil \right] = \\
&= 2 \cdot \sum_{m=1}^{n-1} \left[\left(L + \frac{8}{B} |H_{\boldsymbol{\ell}}|\right) \cdot \sum_{\|\boldsymbol{\ell}\|_1 = m+d-1} \lceil \log_2 |CG(H_{\boldsymbol{\ell}})| \rceil \right] = \\
&= 2 \cdot \sum_{m=1}^{n-1} \left[\left(L + \frac{8}{B} \cdot 2^{m-1}\right) \cdot A_m^d \cdot \left\lceil \log_2 \left(\sum_{i=1}^{\min(d,n-m+1)} A_{n-(m-1)-(i-1)}^d \right) \right\rceil \right] = \\
&= 2 \cdot \sum_{m=1}^{n-1} \left[\left(L + \frac{8}{B} \cdot 2^{m-1}\right) \cdot \binom{m+d-2}{d-1} \cdot \left\lceil \log_2 \left(\sum_{i=1}^{\min(d,n-m+1)} \binom{n+d-m-i}{d-1} \right) \right\rceil \right].
\end{aligned} \tag{14}$$

6.2. Simplifying the Formulas

To get a rough idea of the complexities of *Sparse Grid Reduce* and *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,

$$\binom{k}{d} \leq \frac{k^d}{d!} \,. \tag{15}$$

Using (13) as starting point (assuming $n \geq d$), the runtime of *Sparse Grid Reduce* is bounded by

$$SG\ Reduce(n,d) \overset{(12),(15)}{\in} 2 \cdot d \cdot \lceil \log_2(n+d) \rceil \cdot \left(L + \frac{8}{B} \cdot 2^{n-1} \cdot \left(\frac{n^{d-1}}{(d-1)!} + \mathscr{O}\left(n^{d-2}\right) \right) \right) \,. \tag{16}$$

To estimate the runtime of *Subspace Reduce* we need two bounds,

$$\sum_{m=1}^{n-1} (m+d-2)^{d-1} \leq \int_1^n (m+d-2)^{d-1} \,\mathrm{d}m = \left. \frac{(m+d-2)^d}{d} \right|_{m=1}^{n} \leq \frac{(n+d-2)^d}{d} \tag{17}$$

and

$$\sum_{m=1}^{n-1} (m+d-2)^{d-1} \cdot 2^{m-1} \leq \int_1^n (m+d-2)^{d-1} \cdot 2^{m-1} \,\mathrm{d}m \leq (n+d-2)^{d-1} \cdot 2^{n-1} \,. \tag{18}$$

The second estimate is very rough and is used in estimating the bandwidth term of *Subspace Reduce* which is hence overestimated.

Using (14) as starting point, the runtime of *Subspace Reduce* can be bounded by

$$Subspace\ Reduce(n,d) \overset{(15),(17),(18)}{\leq} 2 \cdot \lceil \log_2(n+d) \rceil \cdot \frac{(n+d)^{d-1}}{(d-2)!} \cdot \left(L \cdot \frac{n+d}{d} + \frac{8}{B} \cdot 2^{n-1} \right) \,. \tag{19}$$

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* (16) and *Subspace Reduce* (19) yields

$$\frac{\text{Number of rounds of } Sparse\ Grid\ Reduce(n,d)}{\text{Number of rounds of } Subspace\ Reduce(n,d)} \approx \frac{2 \cdot d \cdot \lceil \log_2(n+d) \rceil}{2 \cdot \lceil \log_2(n+d) \rceil \cdot \frac{(n+d)^d}{(d-2)! \cdot d}} \approx \frac{d!}{(n+d)^d} > \frac{1}{\binom{n}{d}} \,. \tag{20}$$

Hence the number of rounds is significantly larger using *Subspace Reduce*. For the makespan volume, the ratio is roughly

$$\begin{aligned} &\frac{\text{Makespan volume of } Sparse\ Grid\ Reduce(n,d)}{\text{Makespan volume of } Subspace\ Reduce(n,d)} \approx \\ &\qquad \approx \frac{2 \cdot 8 \cdot d \cdot \lceil \log_2(n+d) \rceil \cdot 2^{n-1} \cdot \frac{n^{d-1}}{(d-1)!}}{2 \cdot 8 \cdot \lceil \log_2(n+d) \rceil \cdot \frac{(n+d)^{d-1}}{(d-2)!} \cdot 2^{n-1}} = \frac{d}{d-1} \cdot \frac{n^{d-1}}{(n+d)^{d-1}} \approx \frac{n^{d-1}}{(n+d)^{d-1}} \overset{n \to \infty}{\longrightarrow} 1 \,. \end{aligned} \tag{21}$$

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 (18). The estimate (21) 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 (13) and (14) 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.

6.3. Makespan Volume and Rounds of Parallel Subspace Reduce

A communication-time formula for *Parallel Subspace Reduce* derives easily from the makespan volume and the number of rounds. The other considered cost measures are the same as for *Subspace Reduce*. We formulate this for the *AllReduce*-routine implemented using binomial trees, as reflected by a factor of $\log_2(\cdot)$. If the *AllReduce*-routine is implemented with linear makespan volume, these factors just disappear, the analysis gets simpler and the dependence on d better, but the asymptotic behaviour (ignoring d) is the same.

This analysis is driven by the observation that with decreasing level-sum the subspaces are getting smaller geometrically, but it has to account for that there is less parallelism. The following upper bound on the makespan volume has as factors the size of the subspace, the necessary serialization, and the cost of the reduce:

$$\text{Makespan volume of } \textit{Parallel Subspace Reduce}(n,d) \leq \sum_{a=1}^{n} 2^{n-a} \cdot a^{d-1} \cdot \log_2(a^d) \; .$$

This sum is geometrically decreasing for $a \geq 3d+3$: for $a \geq d$ we have $\log_2(a^d) = d \log_2 a \leq a^2$ such that we can bound $a^{d-1} \cdot \log_2(a^d) \leq a^{d+1}$. Using the well known $\binom{d}{i} \leq d^i$ and the assumed $d+1 \leq a/3$ we get

$$(a+1)^{d+1} = \sum_{i=0}^{d+1} \binom{d+1}{i} a^{d+1-i} \leq \sum_{i=0}^{d+1} (d+1)^i a^{d+1-i} \leq a^{d+1} \sum_{i=0}^{d+1} (1/3)^i < \frac{3}{2} a^{d+1} \; .$$

Hence for $a \geq 3d+3$, the upper bounding summands $s_a = 2^{n-a} a^{d+1}$ fulfill $s_{a+1} \leq s_a \cdot \frac{1}{2} \cdot \frac{3}{2} = \frac{3}{4} s_a$, and we have $\sum_{a=3d+3}^{\infty} s_a \leq s_{3d+3} \cdot 4$. It remains to estimate the first $3d+3$ summands,

$$\sum_{a=1}^{3d+3} 2^{n-a} \cdot a^{d-1} \cdot \log_2(a^d) \leq (3d+3) 2^{n-1} (3d+3)^{d-1} d \log_2(3d+3) \in \mathcal{O}\left(2^{n-1}\right) \; ,$$

such that we can conclude:

$$\text{Makespan volume of } \textit{Parallel Subspace Reduce}(n,d) \in \mathcal{O}\left(2^{n-1}\right) + 4 s_{3d+3} = \mathcal{O}\left(2^{n-1}\right) .$$

This is clearly asymptotically optimal because at least one subspace of size 2^{n-1} needs to be communicated (see also Section 5.3). Note that this is asymptotically less than *Sparse Grid Reduce* (16) and *Subspace Reduce* (19) by a factor of at least n^{d-1}.

To upper bound the number of communication rounds we get the following:

$$\text{Number of rounds of } \textit{Parallel Subspace Reduce}(n,d) \leq \sum_{a=1}^{n} a^{d-1} \cdot \log_2(a^d) \in \mathcal{O}\left(n^d \log_2 n\right) \; .$$

This should be compared with the $\mathcal{O}(\log_2 n)$ rounds of *Sparse Grid Reduce* and is asymptotically the same as that of *Subspace Reduce* as in (19).

Because the above are only asymptotic estimates, and as it is easy to compute these communication parameters as a side product of our implementation (see Section 7.4), we present and compare in Section 8 the number of rounds and the makespan volume for *Sparse Grid Reduce*, *Subspace Reduce* and *Parallel Subspace Reduce* for specific dimensions d and sparse grid levels n.

For the cases with boundary points or minimal level we did not see that the explicit formulas give interesting insights. For high refinement levels and boundary or constant minimal level, the asymptotics remain unchanged. However, for non-constant minimum level it is unclear how to phrase the asymptotic considerations.

7. Testbed and Experimental Setup

This section introduces the systems used for the experiments and states the latency and bandwidth values for each of the systems. It describes the implementations of *Sparse Grid Reduce*, *Subspace Reduce* and *Parallel Subspace Reduce*, as well as the modifications done to *Subspace Reduce* such that non-blocking MPI routines can be used. Furthermore, it elaborates on how the communication model is applied to predict the runtime of the communication algorithms.

Table 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.

	Lower Bound		Upper Bound	
	L_0	B_0	L_1	B_1
Hermit	1.4 μs	6.0 GByte/s	9 μs	1.0 GByte/s
SuperMUC	2.0 μs	4.7 GByte/s	4 μs	1.0 GByte/s

	Lower Bound	
	L_0	B_0
Stampede	1.1 μs	6.4 GByte/s
JUQUEEN	1.7 μs	1.8 GByte/s

7.1. Systems used for Measurements

We have performed our experiments on two Tier-0/1 systems, *Hermit* and *SuperMUC*. To obtain reliable data for our communication model, we measured the latency and bandwidth for MPI messages on both systems with the so-called ping-pong test. This 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 the message back to node 1. This procedure is performed for different message sizes and 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 2. Furthermore, these values were used for the estimated runtimes presented later in the Tables 4, 5, 6, 7 and 8.

Obtaining upper bounds for the model, it has to be considered that the measured bandwidth typically increases with growing message size. The effect that only a fraction of the full bandwidth were measured 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*.

7.1.1. Hermit

Hermit is a Cray XE6 system located at the High Performance Computing Center Stuttgart (HLRS) with a reported LINPACK performance of 831.4 TFlops/s. It consists of 3552 nodes connected in a 3-dimensional torus network of Cray Gemini interconnects. Each node has 32 GByte DDR3 main memory and is equipped with two AMD Interlagos CPUs, which have 16 cores and 16 MByte L3 Cache each. 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 *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 7.3.2 for 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 μs and 8 μs and a maximum bandwidth between 1.7 GByte/s and 6 GByte/s depending on the position of the nodes in the network. This is in agreement with material presented by HLRS [54]. For Hermit we have chosen the following values for the model: lower bound: $L_0 = 1.4$ μs, $B_0 = 6$ GByte/s; upper bound: $L_1 = 9$ μs, $B_1 = 1$ GByte/s.

7.1.2. SuperMUC

SuperMUC, which is located at the Leibniz-Rechenzentrum (LRZ), currently is the second fastest system in Germany with a LINPACK performance of 2897.0 TFlops/s. 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. Within each island the nodes are connected by a fully non-blocking Infiniband 4xFDR10 network. The islands are connected to each other by 126 spine switches. Each thin node has 32 GB DDR3 RAM and two Intel Xeon E5-2680 (Sandy Bridge) CPUs with 8 cores and 20 MB L3 cache each. 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 simultaneously communicate at the same bandwidth and latency. Therefore, we only used the nodes within a single island to decrease the variance of our measurements. Sending messages inside an island, we have measured a minimum latency of 2 μs and a maximum bandwidth of 4.7 GByte/s, which agrees with data found by other researchers. For SuperMUC we have determined the following values for the model: lower bound: $L_0 = 2$ μs, $B_0 = 4.7$ GByte/s; upper bound: $L_1 = 4$ μs, $B_1 = 1$ GByte/s.

7.2. Additional Systems Employed to Predict Runtimes

We extend the predictions of our model to other Top500 systems with a different interconnect in Table 8, namely *JUQUEEN* and *Stampede*. We did not have access to these systems, so we can only rely on data provided by training material of the compute centers.

7.2.1. 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 reported latency is between 1.1 μs and 2.54 μs depending on the number of switch hops. The maximum bandwidth peaks below 6.4 GByte/s [55]. The values $L_0 = 1.1$ μs and $B_0 = 6.4$ GByte/s were used for the model.

7.2.2. 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 neighbors is 1.7 μs and the maximum bandwidth is 1.8 GByte/s [56].

7.3. 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 $H_{\boldsymbol{\ell}'}$ of the component grid $C_{\boldsymbol{\ell}}$ of its node. The coefficients of each $H_{\boldsymbol{\ell}'}$ are stored in a distinct array. Whenever experiments are conducted for sparse grids with boundary, the boundary is of level-1 type (see Section 5.7.1).

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 $\mathscr{C}_n^d$ 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 sparse grid level n at which the experiments start as well as the minimum-level $\boldsymbol{\ell}_{\min}$ for this n. When increasing the sparse grid level n by one, we increase $\boldsymbol{\ell}_{\min}$ component-wise in a round robin fashion. Thus, $\|\boldsymbol{\ell}_{\min}\|_1$ is also increased by one. For example: for $d = 5$, we started with $n = 7$ and $\boldsymbol{\ell}_{\min} = (1,1,1,1,1)$. Increasing n means setting $\boldsymbol{\ell}_{\min} = (1,1,1,1,2)$ for $n = 8$, $\boldsymbol{\ell}_{\min} = (1,1,1,2,2)$ for $n = 9$, $\boldsymbol{\ell}_{\min} = (2,2,2,2,2)$ for $n = 12$, and so on. All those cases result in 456 component grids. 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.

In order to avoid large variance in our measurements due external influences like the overall load on the system's communication network, we report average times for our experiments. Each of the experiments has been repeated at least three times and until at least 0.1 seconds have elapsed. When repeating an experiment the same set of communication nodes has been employed. For the experiments with constant $\boldsymbol{\ell}_{\min}$, a different set of nodes on the system has been used for different number of component grids. For the experiments, where the minimum-level has been increased, and hence the number of component grids and communication nodes remained constant, always the same set of nodes had been used on SuperMUC. For Hermit, the same set of nodes was used for $d = 5$, $n = 7$ and $\boldsymbol{\ell}_{\min} = (1,1,1,1,1)$ 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 $\boldsymbol{\ell}_{\min} = (1,1,1,1,1)$ (126 nodes) and for $d = 10$, $n = 4$ and $\boldsymbol{\ell}_{\min} = (1,1,1,1,1,1,1,1,1,1)$ (286 nodes), different sets of compute nodes were used for each communication task as provided by Hermit's scheduling system.

7.3.1. Sparse Grid Reduce

Sparse Grid Reduce, as shown in Algorithm 1, is performed in three steps. First, the coefficients of all $H_{\mathbf{k}} \subset \mathrm{SG}^d_{n-1}$ are copied into a single buffer. For the $H_{\mathbf{k}}$ which do not exist in the component grid $C_{\boldsymbol{\ell}}$ of a particular process, i.e $H_{\mathbf{k}} \nsubseteq C_{\boldsymbol{\ell}}$, the buffer is filled with zeros. Then, the *MPI_Allreduce* function is executed on all MPI processes in order to perform the *AllReduce* on the buffer. As a last step, the (combined) coefficients are copied from the buffer to their corresponding $H_{\mathbf{k}}$. For our experiments the runtime of *Sparse Grid Reduce* includes all three steps.

7.3.2. Subspace Reduce

The implementation of *Subspace Reduce* (Algorithm 2) requires an initialization phase. In the initialization phase an MPI Communicator is created for each increment space $H_{\mathbf{k}} \in \mathscr{H}^d_{n-1}$. The MPI Communicator for the increment space $H_{\mathbf{k}}$ contains all compute nodes that contain this increment space, i.e. all nodes $\boldsymbol{\ell}$ for which $H_{\mathbf{k}} \subseteq C_{\boldsymbol{\ell}}$ holds. Thus, the size of the communicators ranges from $d+1$ to $\left|\mathscr{C}^d_n\right|$. 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. For practical application, in most cases the communicators need to be created only once in the beginning and can be reused for many communication steps. In certain circumstances it might, be necessary change the allocation of component grids to the nodes, for example to deal with load balancing or resilience issues. However, it is valid to assume that this happens only rarely and so the communicators can be reused many times. For this reason, we neglected the time for the initialization phase in the measurements.

When performing *Subspace Reduce*, each process loops over all $H_{\mathbf{k}} \in \mathscr{H}^d_{n-1}$ 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, there might be some degree of parallelism, even with the "serial" *Subspace Reduce*. For *Parallel Subspace Reduce*, the increment spaces have been reordered according to Section 5.6 (during the initialization phase) to allow a parallel execution.

For both versions of *Subspace Reduce*: If the increment space $H_{\mathbf{k}}$ exists on the process, the coefficients of $H_{\mathbf{k}}$ are copied to the buffer of the process. Then, *MPI_Allreduce* is executed on all processes included in the communicator of $H_{\mathbf{k}}$ in order to perform *AllReduce* on the buffered coefficients. Afterwards, the (combined) coefficients are copied back from the buffer to $H_{\mathbf{k}}$. When an increment space $H_{\mathbf{k}}$ 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 $H_{\mathbf{k}} \in \mathscr{H}^d_{n-1}$ and include the two copy operations.

Since the MPI implementation by Cray available on Hermit offers non-blocking collective operations, we additionally use a modified version of *Subspace Reduce* employing the non-blocking *MPI_Iallreduce*-routine. Using the non-blocking routines *Subspace Reduce* is implemented as follows. First, each process executes a loop over all $H_{\mathbf{k}} \in \mathscr{H}^d_{n-1}$. If the increment space $H_{\mathbf{k}}$ 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 $H_{\mathbf{k}}$. After the loop, we execute the *MPI_Waitall* function in order to wait until all reduce operations have been executed. Afterwards, each process loops over all $H_{\mathbf{k}} \in \mathscr{H}^d_{n-1}$ and extracts the (combined) coefficients of increment spaces $H_{\mathbf{k}}$ that are contained in the component grid $C_{\boldsymbol{\ell}}$ of the process.

7.4. Exact Modeling of the Communication Times

In Section 6.1, exact formulas for the number of rounds and the makespan volume of *Sparse Grid Reduce* (13) and *Subspace Reduce* (14) (assuming all increment spaces are reduced strictly in series for the latter) 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 in 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^d_{n-1} and the number of component grids in $\mathscr{C}^d_n$. For *Subspace Reduce* we need the level vector of each hierarchical increment space $H_{\boldsymbol{\ell}} \in \mathscr{H}^d_{n-1}$, 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

Table 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. $\boldsymbol{\ell}_{\min} = \mathbf{1}$.

$\mathbf{d} = 3$	Nbr. of Comp. Grids
$n = 5$	31
$n = 10$	136
$n = 15$	316
$n = 20$	571

$\mathbf{d} = 5$	Nbr. of Comp. Grids
$n = 5$	126
$n = 10$	1,876
$n = 15$	9,626
$n = 20$	30,876

$\mathbf{d} = 10$	Nbr. of Comp. Grids
$n = 4$	286
$n = 8$	19,448
$n = 12$	352,705

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 5. Note that some modern MPI implementations do not distinguish between reduce and broadcast steps and use more sophisticated approaches. This can then lead to an even smaller total number of rounds and makespan volume.

8. Simulations and Experiments

This section presents the experimental results. First, we compare the number of rounds, the makespan volume and hence the predicted runtime of the different algorithms to each other. Then we present experiments on Hermit and SuperMUC which we back up with bounds obtained from the communication model. We conclude with an outlook on the communication times of problems that would be out of scope for experiments.

In summary, we have observed that *Sparse Grid Reduce* performs best for small levels and dimensions. Here, the overhead of communicating the whole sparse grid is small and communicating in few rounds is advantageous. If either the dimension or the level grows, *Subspace Reduce* outperforms *Sparse Grid Reduce* as significantly less data is communicated. The reordering of the increment spaces enables *Parallel Subspace Reduce* to further improve communication time when blocking MPI-routines are used. On Hermit, where non-blocking collective communication was available, the performance of *Subspace Reduce* and *Parallel Subspace Reduce* could be further improved. In particular, with the non-blocking routines the runtime of *Subspace Reduce* and *Parallel Subspace Reduce* is very similar. If the number of increment spaces and component grids grows, the speedup of *Parallel Subspace Reduce* increases with increasing level. For a constant number of increment spaces and component grids, the performance benefit of *Parallel Subspace Reduce* over *Subspace Reduce* is limited.

8.1. Predicting Runtimes

This section analyzes the number of rounds and the makespan volume for *Sparse Grid Reduce* and the original and parallel version of *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 actual data representation (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 used real-valued floating point numbers with double precision (8 bytes per grid point).

Table 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. $\boldsymbol{\ell}_{\min} = \mathbf{1}$. For $d = 5$ and $d = 10$ the number of component grids quickly exceeds the number of nodes available on Hermit or SuperMUC. Here, for moderate to high levels we have to limit ourselves to theoretical results. For experiments with $d = 5$ and $d = 10$ we need to employ an increasing minimum level to limit the number of required component grids and nodes. Still, we will rediscover the most important trends of this subsection in the actual experiments. Since the experimental results with constant minimum level ($d = 3$), as well as with increasing minimum level ($d = 5$ and $d = 10$), will fit very well to the model bounds, we are confident that the additional predictions are good estimates even for the runtimes of the experiments that are beyond our resource limitations.

Table 4 shows the number of rounds and the makespan volume for sparse grids without boundary and without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$). Furthermore, we depict the ratio between *Sparse Grid Reduce* and the original and

Table 4: Without boundary and without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$): rounds and makespan volume (MkVol) for different dimensions. The makespan volume is expressed in the number of grid points (not bytes).

$\mathbf{d = 3}$	*Sparse Grid Reduce*		*Subspace Reduce*		*Par. Subspace Reduce*		$\frac{\textit{Sparse Grid Reduce}}{\textit{Subspace Reduce}}$		$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$	
	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol
$n = 5$	**10**	1110	128	**582**	104	**390**	0.078	1.9	0.096	2.8
$n = 10$	**16**	303088	1400	**111860**	880	**33908**	0.011	2.7	0.018	8.9
$n = 15$	**18**	2.71e+07	5588	**9.28e+06**	3186	**1.45e+06**	0.0032	2.9	0.0056	19
$n = 20$	**20**	1.8e+09	14836	**5.67e+08**	8016	**5.14e+07**	0.0013	3.2	0.0025	35

$\mathbf{d = 5}$	*Sparse Grid Reduce*		*Subspace Reduce*		*Par. Subspace Reduce*		$\frac{\textit{Sparse Grid Reduce}}{\textit{Subspace Reduce}}$		$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$	
	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol
$n = 5$	**14**	4914	434	**2414**	314	**1454**	0.032	2	0.045	3.4
$n = 10$	**22**	4.13e+06	12736	**1.42e+06**	6586	**321234**	0.0017	2.9	0.0033	13
$n = 15$	**28**	8.77e+08	100284	**2.49e+08**	45818	**2.57e+07**	0.00028	3.5	0.00061	34
$n = 20$	**30**	9.68e+10	446574	**2.63e+10**	190776	**1.37e+09**	6.7e-05	3.7	0.00016	71

$\mathbf{d = 10}$	*Sparse Grid Reduce*		*Subspace Reduce*		*Par. Subspace Reduce*		$\frac{\textit{Sparse Grid Reduce}}{\textit{Subspace Reduce}}$		$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$	
	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol
$n = 4$	**18**	4338	598	**2058**	526	**1770**	0.03	2.1	0.034	2.5
$n = 8$	**30**	1.19e+07	86948	**3.71e+06**	43544	**1.16e+06**	0.00035	3.2	0.00069	10
$n = 12$	**38**	4.85e+09	2.29e+06	**1.26e+09**	907862	**2.06e+08**	1.7e-05	3.9	4.2e-05	24

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 *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. 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. For $d = 5$ and $n = 20$, the communication 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*. 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 about 11 for $d = 3$, $n = 20$, by a factor of about 19 for $d = 5$, $n = 20$ and a factor of about 6 for $d = 10$, $n = 12$, compared to *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 makespan volume for *Parallel Subspace Reduce*.

Table 5 compares the number of rounds and the makespan volume of sparse grids with boundary points for $d = 5$. With boundary points the size of the sparse grid is drastically increased. The number of increment spaces and the whole structure of the communication task, however, is 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 without boundary points. The makespan volume increases for all methods due to the increased volume of the increment spaces containing the boundary points. Especially for small levels, the makespan volume increases by several orders of magnitude. Comparing the ratio of the makespan volume of *Sparse Grid Reduce* and *Subspace Reduce* with and without boundary, it can be seen that the increase in makespan volume is very similar for both methods. In contrast, the makespan volume of *Parallel Subspace Reduce* increases faster than the makespan volume of the other two algorithms. For $d = 5$ the ratio

Table 5: With boundary and without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$): rounds and makespan volume (MkVol) for $d = 5$. The makespan volume is expressed in the number of grid points (not bytes).

$\mathbf{d = 5}$	*Sparse Grid Reduce*		*Subspace Reduce*		*Par. Subspace Reduce*		$\frac{\textit{Sparse Grid Reduce}}{\textit{Subspace Reduce}}$		$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$	
	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol
$n = 5$	**14**	168462	434	**89802**	314	**63882**	0.032	1.9	0.045	2.6
$n = 10$	**22**	4.33e+07	12736	**1.57e+07**	6586	**4.61e+06**	0.0017	2.8	0.0033	9.4
$n = 15$	**28**	5.23e+09	100284	**1.53e+09**	45818	**2.22e+08**	0.00028	3.4	0.00061	24
$n = 20$	**30**	4.11e+11	446574	**1.14e+11**	190776	**8.61e+09**	6.7e-05	3.6	0.00016	48

Table 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).

$\mathbf{d = 5}$	*Sparse Grid Reduce*		*Subspace Reduce*		*Par. Subspace Reduce*		$\frac{\textit{Sparse Grid Reduce}}{\textit{Subspace Reduce}}$		$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$	
456 nodes	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol	Rounds	MkVol
$n = 7$	**18**	1.85e+06	2184	**795006**	1324	**403326**	0.0082	2.3	0.014	4.6
$n = 10$	**18**	1.08e+07	2184	**4.59e+06**	1324	**2.25e+06**	0.0082	2.3	0.014	4.8
$n = 15$	**18**	2.32e+08	2184	**9.8e+07**	1324	**4.61e+07**	0.0082	2.4	0.014	5
$n = 20$	**18**	5.97e+09	2184	**2.51e+09**	1324	**1.15e+09**	0.0082	2.4	0.014	5.2

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 4). This is due to the fact that increment spaces with a small level sum contain many boundary points in comparison to interior points and also contribute to many component grids. For these increment spaces the volume increases most, but they 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. However, they contain only relatively few boundary points in comparison to interior points.

Table 6 presents the case of increasing minimum-level for $d = 5$ with boundary points. The minimum-level is increased as described in Section 7.3 so that a constant number of 456 component grids are used for each level n. We started with $\boldsymbol{\ell}_{\min} = (1,1,1,1,1)$ 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. Like before, the number of rounds for *Sparse Grid Reduce* is significantly smaller than for *Subspace Reduce* and *Parallel Subspace Reduce*, 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 $\boldsymbol{\ell}_{\min}$ (see Table 5) where this ratio increases significantly when n grows, in particular for *Parallel Subspace Reduce*. When using a minimum level, the communication task is much smaller than in the case without minimum level. Furthermore, if the minimum level is increased simultaneously with the level n, the number of hierarchical increment spaces remains constant and the structure of the communication task remains unchanged. Although the level vectors $\boldsymbol{\ell}$ of the increment spaces increase when n grows, the differences $\boldsymbol{\ell} - \boldsymbol{\ell}_{\min}$ remain unchanged. This relative level $\boldsymbol{\ell} - \boldsymbol{\ell}_{\min}$, however, determines which increment spaces can be reduced in parallel. Thus, the amount of parallelism does not grow with n, as in the case without minimum level. All in all, with *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, nearly constant speedups in the experiments.

8.2. Performance Prediction

In this section we limit the discussion to *Sparse Grid Reduce* and *Parallel Subspace Reduce* and disregard *Subspace Reduce*. Table 7 shows the predicted runtime of *Sparse Grid Reduce* and *Parallel Subspace Reduce* on Hermit, using the lower bound values L_0 and B_0, as well as the fraction of time spent in the latency and bandwidth terms. This experiment was conducted with boundary points and no minimum-level. For both methods the percentage of time spent in the bandwidth term grows with increasing level and dimension. However, for *Sparse Grid Reduce* it starts off

Table 7: With boundary and without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$): total predicted runtime split into latency and bandwidth term for Hermit ($L_0 = 1.4\ \mu$s, $B_0 = 6$ GByte/s) for different dimensions.

$\mathbf{d = 3}$	*Sparse Grid Reduce*			*Par. Subspace Reduce*			$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$
	Latency	Bandwidth	Total	Latency	Bandwidth	Total	
$n = 5$	63.9 %	36.1 %	**0.0000 s**	97.4 %	2.6 %	0.0001 s	0.15
$n = 10$	2.1 %	97.9 %	**0.0011 s**	88.0 %	12.0 %	0.0014 s	0.77
$n = 15$	0.0 %	100.0 %	0.072 s	42.0 %	58.0 %	**0.011 s**	6.7
$n = 20$	0.0 %	100.0 %	4.1 s	5.4 %	94.6 %	**0.21 s**	20

$\mathbf{d = 5}$	*Sparse Grid Reduce*			*Par. Subspace Reduce*			$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$
	Latency	Bandwidth	Total	Latency	Bandwidth	Total	
$n = 5$	8.0 %	92.0 %	**0.0002 s**	83.8 %	16.2 %	0.0005 s	0.47
$n = 10$	0.1 %	99.9 %	0.058 s	60.0 %	40.0 %	**0.015 s**	3.8
$n = 15$	0.0 %	100.0 %	7.0 s	17.8 %	82.2 %	**0.36 s**	19
$n = 20$	0.0 %	100.0 %	548 s	2.3 %	97.7 %	**12 s**	47

$\mathbf{d = 10}$	*Sparse Grid Reduce*			*Par. Subspace Reduce*			$\frac{\textit{Sparse Grid Reduce}}{\textit{Par. Subspace Reduce}}$
	Latency	Bandwidth	Total	Latency	Bandwidth	Total	
$n = 4$	0.0 %	100.0 %	0.058 s	3.2 %	96.8 %	**0.023 s**	2.5
$n = 8$	0.0 %	100.0 %	22 s	2.4 %	97.6 %	**2.6 s**	8.6
$n = 12$	0.0 %	100.0 %	2745 s	1.0 %	99.0 %	**130 s**	21

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. *Parallel 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 *Sparse Grid Reduce* is faster for a small to medium dimension and small level, *Parallel Subspace Reduce* outperforms it in all other cases. In the first case, the volume overhead by *Sparse Grid Reduce* is not too large. Here, it pays off that *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, so that *Parallel Subspace Reduce* is predicted to outperform *Sparse Grid Reduce* by one to two orders of magnitude.

Table 8 compares the predicted runtimes of *Sparse Grid Reduce* and *Parallel Subspace Reduce* on the four systems presented in Section 7 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 *Sparse Grid Reduce* and *Parallel Subspace Reduce* only the ratio between latency and bandwidth is important, but not the actual 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 *Sparse Grid Reduce* and *Parallel Subspace Reduce* is very similar. On all four systems, the table shows that *Parallel Subspace Reduce* is expected to outperform *Sparse Grid Reduce* for $n \geq 10$. For $n = 20$, *Parallel Subspace Reduce* is expected to be about 47 times faster than *Sparse Grid Reduce*. Only when level and makespan volume are small ($n = 5$), *Sparse Grid Reduce* is expected to perform best. If the latency of Hermit would be reduced by a factor of 10, *Parallel Subspace Reduce* would be faster than *Sparse Grid Reduce* even for $n = 5$. For small n, where a large fraction of the total time of *Parallel Subspace Reduce* is spent in the latency term (see Table 7), *Parallel Subspace Reduce* benefits from the reduced latency. In contrast, increasing the latency by a factor of 10 increases the runtime for *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 *Sparse Grid Reduce*. Here, the fraction of time consumed by the latency term is negligible and only the bandwidth term is relevant for medium to high dimensions and levels (see Table 7).

Table 8: With boundary and without minimum level (i.e. $\ell_{\min} = \mathbf{1}$): runtime predictions for different systems and $d = 5$. Model parameters: Hermit ($L_0 = 1.4\ \mu$s, $B_0 = 6$ GByte/s), SuperMUC ($L_0 = 2\ \mu$s, $B_0 = 4.7$ GByte/s), Stampede ($L_0 = 1.1\ \mu$s, $B_0 = 6.4$ GByte/s), JUQUEEN ($L_0 = 1.7\ \mu$s, $B_0 = 1.8$ GByte/s).

d = 5	**Hermit** $L = L_0,\ B = B_0$			**Hermit Latency decreased** $L = L_0/10,\ B = B_0$			**Hermit Latency increased** $L = 10 \cdot L_0,\ B = B_0$		
	SG Red.	*P. Subsp. Red.*	$\frac{\text{SG Red.}}{\text{P. Subsp. Red.}}$	*SG Red.*	*P. Subsp. Red.*	$\frac{\text{SG Red.}}{\text{P. Subsp. Red.}}$	*SG Red.*	*P. Subsp. Red.*	$\frac{\text{SG Red.}}{\text{P. Subsp. Red.}}$
$n = 5$	**0.0002 s**	0.0005 s	0.47	0.0002 s	**0.0001 s**	1.8	**0.0004 s**	0.0045 s	0.094
$n = 10$	0.058 s	**0.015 s**	3.8	0.058 s	**0.0071 s**	8.2	**0.058 s**	0.098 s	0.59
$n = 15$	7.0 s	**0.36 s**	19.4	7.0 s	**0.30 s**	23.1	7.0 s	**0.94 s**	7.4
$n = 20$	548 s	**12 s**	46.7	548 s	**12 s**	47.7	548 s	**14 s**	38.8

d = 5	**SuperMUC**			**Stampede**			**JUQUEEN**		
	SG Red.	*P. Subsp. Red.*	$\frac{\text{SG Red.}}{\text{P. Subsp. Red.}}$	*SG Red.*	*P. Subsp. Red.*	$\frac{\text{SG Red.}}{\text{P. Subsp. Red.}}$	*SG Red.*	*P. Subsp. Red.*	$\frac{\text{SG Red.}}{\text{P. Subsp. Red.}}$
$n = 5$	**0.0003 s**	0.0007 s	0.43	**0.0002 s**	0.0004 s	0.53	**0.0008 s**	0.0008 s	0.94
$n = 10$	0.074 s	**0.021 s**	3.5	0.054 s	**0.013 s**	4.2	0.19 s	**0.032 s**	6.1
$n = 15$	8.9 s	**0.47 s**	19.0	6.5 s	**0.33 s**	19.9	23 s	**1.1 s**	21.8
$n = 20$	700 s	**15 s**	46.6	514 s	**11 s**	46.9	1827 s	**39 s**	47.4

8.3. Results on Hermit

This section discusses the experiments conducted on Hermit. Unless explicitly specified, 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 6 (left) shows the results for $d = 3$ without boundary points and a constant minimum level. The number of nodes ranges from 10 to 460. For $n \leq 11$, *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) and so the volume overhead of *Sparse Grid Reduce* is only low. However, with growing n the runtime of *Sparse Grid Reduce* increases faster than for the other methods. For $n = 18$ the total time of *Sparse Grid Reduce* was 1.12s and the total time of *Parallel Subspace Reduce* was 0.10s, which is a factor of 11. *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 higher the degree of parallelism that 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 4. The two non-blocking variants of *Subspace Reduce* only show a slight improvement over blocking *Parallel Subspace Reduce*. From the fact that there is no visible difference between the two non-blocking variants, we conclude that reordering the communication tasks has no performance benefit in case of the non-blocking implementations. An explanation for this behavior would be that the MPI system is able to rearrange the communication on a lower level and with a finer granularity (single messages or even network packets) than we are able to with our algorithms. Later we will also see that the artificial reordering of the subspaces will even have a slight adverse effect on the performance in some cases. Figure 6 (right) shows the results with boundary points for $d = 3$ on 10 to 361 nodes. As in the case without boundary, *Sparse Grid Reduce* is faster than the other methods for small n. However, for $n \leq 9$ the difference between *Sparse Grid Reduce* and *Parallel Subspace Reduce* is considerably smaller than in the case without boundary points. The crossover point between *Sparse Grid Reduce* and *Parallel Subspace Reduce* shifted from $n = 12$, as in the case without boundaries, to $n = 10$. Due to the increased size of the increment spaces containing the boundary points, the volume overhead of *Sparse Grid Reduce* increases faster than for the other methods with growing n. For $n = 16$ the total time of *Sparse Grid Reduce* was $0.46s$ and the total time of *Parallel Subspace Reduce* was $0.059s$. Hence, *Parallel Subspace Reduce* is about 7.8 times faster than *Sparse Grid Reduce*. Again, the non-blocking variants show only a slight improvement over *Parallel Subspace Reduce* and also there is no noticeable difference between the two variants.

Figure 7 (left) shows the results for $d = 5$ and boundary points with a constant number of 126 nodes. We can observe similar behavior as with $d = 3$ and fixed minimum level: for small levels *Sparse Grid Reduce* was slightly

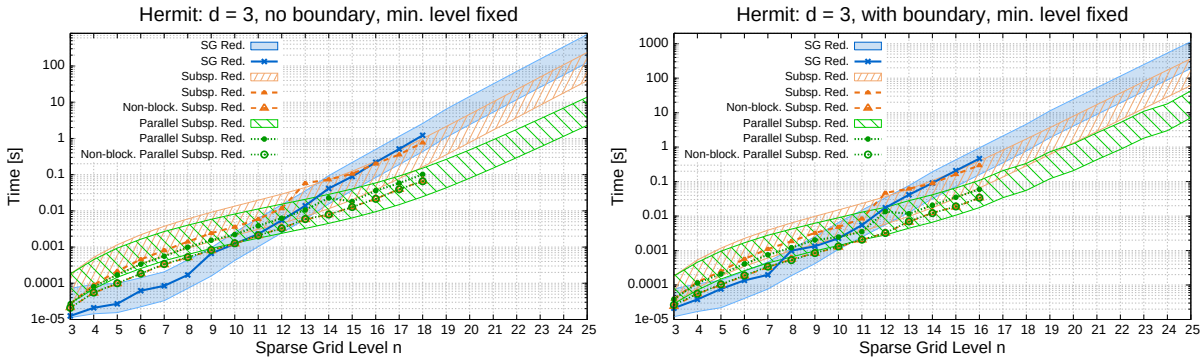

Figure 6: Results on Hermit for $d = 3$ without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$). Hence, the number of nodes increases with n. Left: without boundary, right: with boundary.

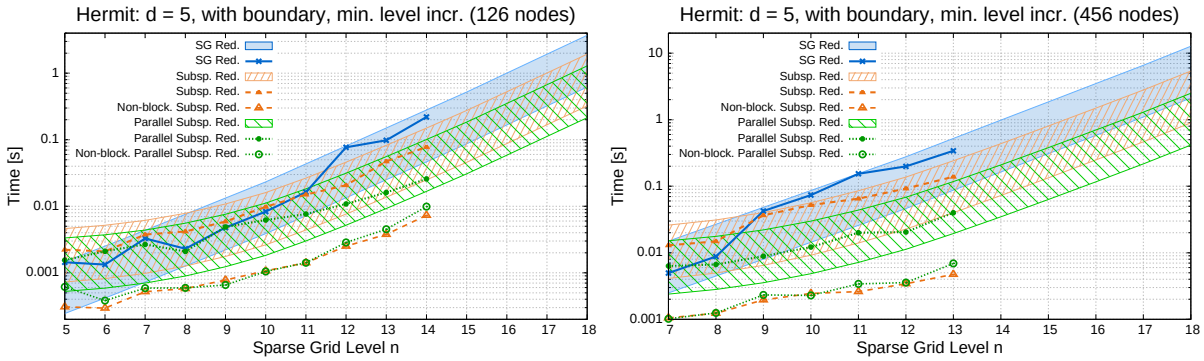

Figure 7: Results on Hermit for $d = 5$ with boundary points. Left: $\boldsymbol{\ell}_{\min}$ increasing from $(1,1,1,1,1)$ for $n = 5$ to $(3,3,3,3,2)$ for $n = 14$, i.e. a constant number of 126 component grids was used. Right: $\boldsymbol{\ell}_{\min}$ increasing from $(1,1,1,1,1)$ for $n = 7$ to $(3,2,2,2,2)$ for $n = 13$, i.e. a constant number of 456 component grids was used.

faster than *Subspace Reduce* or *Parallel Subspace Reduce*, but when n grows, the runtime of *Sparse Grid Reduce* increases faster than the runtime of the other methods. For $n = 14$, the runtime of *Sparse Grid Reduce* was $0.21\,s$, while *Parallel Subspace Reduce* executed in $0.026\,s$, which is a factor of about 8.1. The non-blocking variants of *Subspace Reduce* show a significant improvement over *Parallel Subspace Reduce*. For $n = 14$ the total time of the non-blocking *Subspace Reduce* was 0.0073 s, which is 29 times faster than *Sparse Grid Reduce*. Unlike it was observed for $d = 3$, even for small n the non-blocking variants of *Subspace Reduce* clearly outperform *Sparse Grid Reduce*. Here, however, reordering the increment spaces has a small negative effect on the performance of non-blocking *Parallel Subspace Reduce* for most n. With 456 nodes a very similar behavior can be observed (see Figure 7 (right)). However, the crossover point between *Sparse Grid Reduce* and *Subspace Reduce* shifted from $n = 11$, where it was for 126 nodes, to $n = 9$. For the same n, the number of increment spaces, and thus the total size of the sparse grid, is larger than in the case with 126 nodes, which results in a higher overhead for *Sparse Grid Reduce*. 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. However, unlike for $d = 3$ and a growing number of component grids, the difference between *Subspace Reduce* and *Parallel Subspace Reduce* does not increase with n, because the communication task remains the same. Note that this behavior fits well to the predictions in Table 6, where *Parallel Subspace Reduce* has roughly half the number of rounds and half the makespan volume as *Subspace Reduce* independent of n. For $n = 13$ the total time of non-blocking *Subspace Reduce* was $0.0047\,s$, which is 72 times faster than *Sparse Grid Reduce*.

Figure 8 (left) shows the results for $d = 10$ with boundary points on 286 nodes. Unlike observed for $d = 3$ and

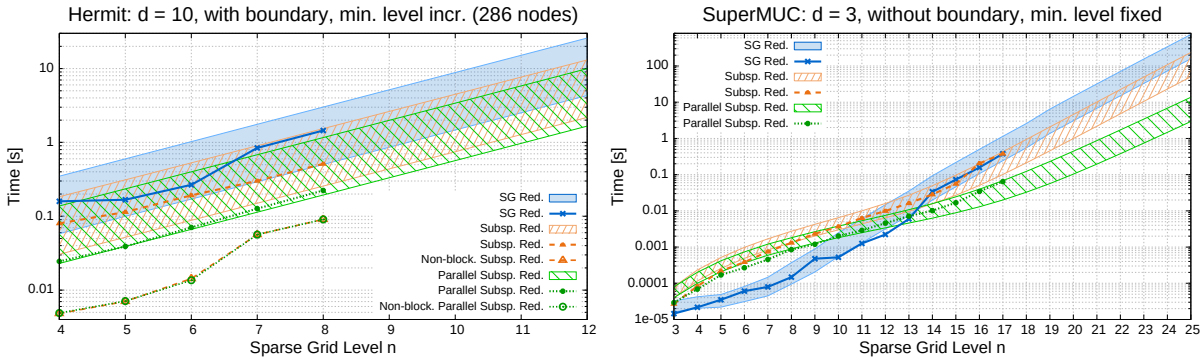

Figure 8: Left: Results on Hermit for $d = 10$ with boundary and $\boldsymbol{\ell}_{\min}$ increasing from $(1,1,1,1,1,1,1,1,1,1)$ for $n = 4$ to $(2,2,2,2,1,1,1,1,1,1)$ for $n = 8$. A constant number of 286 component grids was used. Right: Results on SuperMUC for $d = 3$ without boundary and without minimum level ($\boldsymbol{\ell}_{\min} = \mathbf{1}$). Thus, the number of nodes increases with n.

$d = 5$, *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 for *Sparse Grid Reduce*. 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 of *Parallel Subspace Reduce* over *Subspace Reduce* remains nearly constant with increasing n. 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 non-blocking *Parallel Subspace Reduce* was $0.091\,s$, which is 16 times faster than *Sparse Grid Reduce*.

Though we focus on the optimization of communication schemes in this work and not on the underlying simulations, we provide details about realistic runtimes here for comparison. The experiment in Fig. 7 is closest to the target scenarios we aim for. The simulation of the linear ITG instability with GENE is a $d = 5$ scenario and needs at least the resolution for $n = 12$. There, the compute time for 100 time steps on the largest component grids takes in average $0.07\,s$ on Hermit on one node. 100 time steps is a realistic simulation time span between the global communication steps here (which depends on the simulation task). We were able to reduce the communication time from about $0.19\,s$ (*Sparse Grid Reduce*) to about $0.0034\,s$ (non-blocking *Parallel Subspace Reduce*) for this scenario – a reduction by a factor of 56. Thus, the global communication takes only about as long as 5 time steps of the simulation, which is an excellent result. For other and future simulation scenarios with GENE, the compute time per time step will increase, even with a much higher degree of parallelism per component grid. There, the non-blocking *Parallel Subspace Reduce* will quite likely take less than a single time step, and recombination after each time step might even become feasible.

8.4. Results on SuperMUC

In the following, we present the experimental results on SuperMUC. We could not perform experiments with the non-blocking variants of *Subspace Reduce*, because an MPI implementation which offered the non-blocking collective operations of the MPI 3.0 standard was not yet available when we conducted this research.

Figure 8 (right) shows the results for $d = 3$ without boundary and with constant minimum level on 10 to 409 nodes. We can observe a very similar behavior as on Hermit (see Figure 6). However, the crossover point between *Sparse Grid Reduce* and *Parallel Subspace Reduce* moved from $n = 12$ to $n = 13$. For $n = 17$ the total time of *Sparse Grid Reduce* was 0.38 s and of *Parallel Subspace Reduce* 0.064 s, which is a factor of 5.9. Unlike observed on Hermit, there is no clear trend of *Sparse Grid Reduce* becoming slower than *Subspace Reduce* with increasing n.

Figure 9 (left) shows the results for $d = 5$ with boundary points and a constant number of 456 nodes. The results are very similar to those on Hermit, so please refer to the particular paragraph in Section 8.3 for a detailed discussion. 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. With a factor of 8.5, the gap between the two algorithms is similar on Hermit, but the absolute runtimes were slightly lower on SuperMUC.

Figure 9 (right) shows the results for $d = 10$ with boundary points on and a constant number of 286 nodes. The same trends as on Hermit can be observed. There, we already observed that the runtimes of *Parallel Subspace*

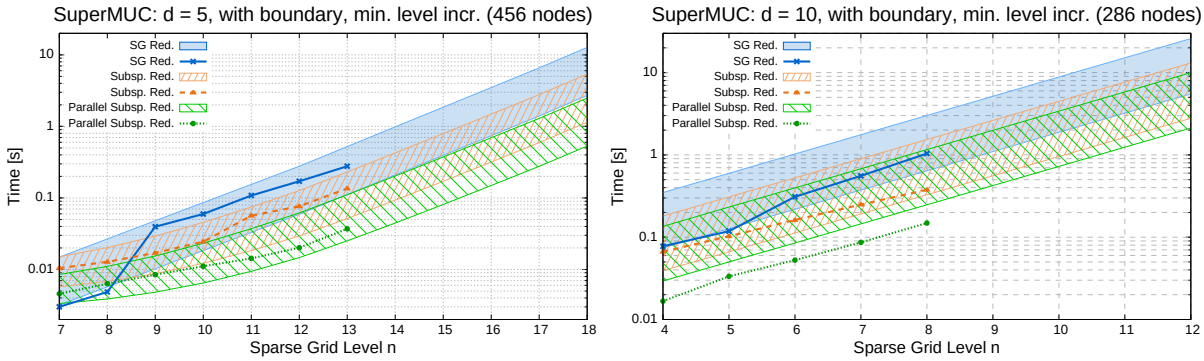

Figure 9: Left: Results on SuperMUC for $d = 5$ with boundary, $\boldsymbol{\ell}_{\min}$ increasing from $(1,1,1,1,1)$ for $n = 7$ to $(3,2,2,2,2)$ for $n = 13$, resulting in a constant number of 426 component grids. Right: Results on SuperMUC for $d = 10$ with boundary, $\boldsymbol{\ell}_{\min}$ increasing from $(1,1,1,1,1,1,1,1,1,1)$ 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.

Reduce come very close to the lower bound of the model. Here, however, they are significantly below. A possible explanation will be given in the next section. 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. On Hermit the factor was 6.6, but the absolute runtimes where more than 46% higher than on SuperMUC.

The experimental results on SuperMUC, as well as on Hermit, agree very well with the model. Apart from a few outliers, the runtimes are inside the prediction intervals and the model reflects the trends of the measurements. Violations of the lower bound might be due to the fact that *MPI_Allreduce* does not necessarily employ an algorithm which uses two phases and binomial trees, as assumed by our model, but instead a more efficient one (see Section 5). We assume that the outliers violating the upper bound on Hermit for $d = 3$ are caused by peculiarities of the MPI implementation or an adverse placement of MPI ranks in the network topology. However, since these cases were so rare, we did not further investigate this issue.

8.5. Extending the Scope to Future Hardware

In this section we finally extend our predictions to 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 $\boldsymbol{\ell}_{\min} = \mathbf{1}$. 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 10 (left) we can see the predicted runtimes for $d = 5$ with boundary and without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$) 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 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 10 (right) shows the predicted runtimes for $d = 10$ with boundary and without minimum level. Here, 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 n to 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 when 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. We expect 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.

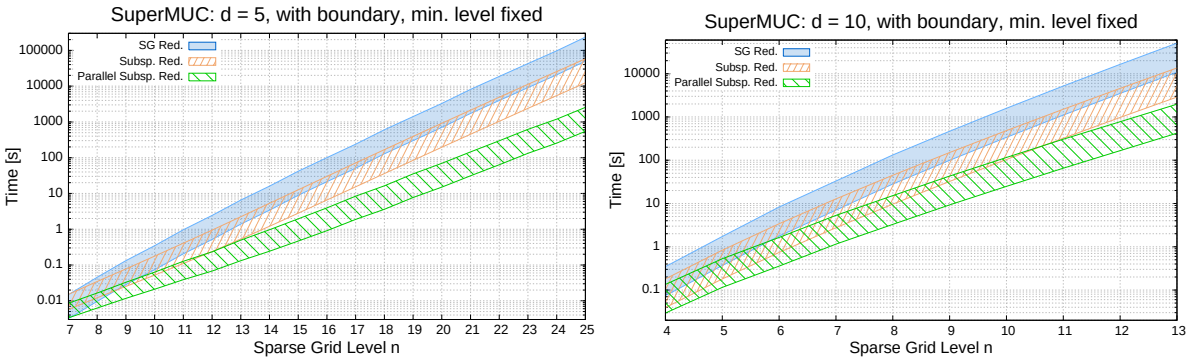

Figure 10: Predictions for SuperMUC with boundary and without minimum level (i.e. $\boldsymbol{\ell}_{\min} = \mathbf{1}$). Thus, the number of component grids increases with n. Left: $d = 5$. Right: $d = 10$.

9. Conclusions and Future Work

The sparse grid 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 anisotropic, 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 paper, 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 to 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 yet and for problem sizes that currently are 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 this even further, up to orders of magnitude depending on the problem size. Nevertheless, there could still be 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. In future work we want to 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 could be explored.

For the experiments, we have used a selection of component grids based on a minimum level. 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.

Acknowledgements

The authors would like to thank Markus Hegland for inspiring initial discussions. We thank the anonymous referees for their valuable comments and for insisting on a careful analysis of *Parallel Subspace Reduce*. This work was partially supported by the German Research Foundation (DFG) through the Priority Programme 1648 Software for Exascale Computing (SPPEXA) and in the Emmy Noether Program, by the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart and by the ”Stiftung der deutschen Wirtschaft”.

References

[1] R. Bellman, Adaptive Control Processes: A Guided Tour, Princeton University Press, 1961.

[2] J. Dongarra, P. H. Beckman, et al., The international exascale software project roadmap, IJHPCA 25 (1) (2011) 3–60.

[3] C. Zenger, Sparse grids, in: W. Hackbusch (Ed.), Parallel Algorithms for Partial Differential Equations, Vol. 31 of Notes on Numerical Fluid Mechanics, Vieweg, 1991, pp. 241–251.
URL http://www5.in.tum.de/pub/zenger91sg.pdf

[4] M. Griebel, M. Schneider, C. Zenger, A combination technique for the solution of sparse grid problems, in: Iterative Methods in Lin. Alg., 1992, pp. 263–281.

[5] M. Hegland, Adaptive sparse grids, in: K. Burrage, R. B. Sidje (Eds.), Proc. of 10th Computational Techniques and Applications Conference CTAC-2001, Vol. 44, 2003, pp. C335–C353, [Online] http://anziamj.austms.org.au/V44/CTAC2001/Hegl [April 1, 2003].

[6] T. Goerler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, D. Told, The global version of the gyrokinetic turbulence code GENE, J. Comput. Phys. 230 (2011) 7053–7071. doi:10.1016/j.jcp.2011.05.034.

[7] C. Kowitz, D. Pflüger, F. Jenko, M. Hegland, The combination technique for the initial value problem in linear gyrokinetics, in: M. Griebel, J. Garcke (Eds.), Sparse Grids and Applications, Vol. 88 of Lecture Notes in Computational Science and Engineering, Springer, Heidelberg, 2012, pp. 205–222.

[8] M. Heene, A. Hinojosa, C. Kowitz, P. Zaspel, T. Dannert, H.-J. Bungartz, M. Griebel, F. Jenko, D. Pflüger, An exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond (EXAHD), in: Proc. of EuroPar, 2014, submitted.

[9] M. Griebel, W. Huber, C. Zenger, Numerical turbulence simulation on a parallel computer using the combination method, in: Flow simulation on high performance computers II, 1996, pp. 34–47.

[10] P. Hupp, R. Jacob, M. Heene, D. Pflüger, M. Hegland, Global communication schemes for the sparse grid combination technique, in: Parallel Computing - Accelerating Computational Science and Engineering (CSE), Vol. 25 of Advances in Parallel Computing, IOS Press, 2014, pp. 564–573.

[11] J.-W. Hong, H. T. Kung, I/O complexity: The red-blue pebble game, in: Proceedings of STOC '81, ACM, New York, NY, USA, 1981, pp. 326–333. doi:http://doi.acm.org/10.1145/800076.802486.

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

[13] A. Aggarwal, A. K. Chandra, M. Snir, Hierarchical memory with block transfer, in: Proceedings of the 28th Annual Symposium on Foundations of Computer Science, SFCS '87, IEEE Computer Society, Washington, DC, USA, 1987, pp. 204–216. doi:10.1109/SFCS.1987.31.
URL http://dx.doi.org/10.1109/SFCS.1987.31

[14] B. Alpern, L. Carter, E. Feig, T. Selker, The uniform memory hierarchy model of computation, Algorithmica 12 (2-3) (1994) 72–109. doi:10.1007/BF01185206.
URL http://dx.doi.org/10.1007/BF01185206

[15] S. Sen, S. Chatterjee, N. Dumir, Towards a theory of cache-efficient algorithms, J. ACM 49 (6) (2002) 828–858. doi:10.1145/602220.602225.
URL http://doi.acm.org/10.1145/602220.602225

[16] L. Arge, M. T. Goodrich, M. Nelson, N. Sitchinava, Fundamental parallel algorithms for private-cache chip multiprocessors, in: Proc. of SPAA '08, ACM.

[17] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS '99, IEEE Computer Society, Washington, DC, USA, 1999, pp. 285–.
URL http://dl.acm.org/citation.cfm?id=795665.796479

[18] L. G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (1990) 103–111. doi:http://doi.acm.org/10.1145/79173.79181.
URL http://doi.acm.org/10.1145/79173.79181

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, Logp: Towards a realistic model of parallel computation, SIGPLAN Not. 28 (7) (1993) 1–12. doi:10.1145/173284.155333.
URL http://doi.acm.org/10.1145/173284.155333

[20] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.
URL http://doi.acm.org/10.1145/1327452.1327492

[21] M. T. Goodrich, N. Sitchinava, Q. Zhang, Sorting, searching, and simulation in the mapreduce framework, in: T. Asano, S.-i. Nakano, Y. Okamoto, O. Watanabe (Eds.), Algorithms and Computation, Vol. 7074 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 374–383. doi:10.1007/978-3-642-25591-5_39.
URL http://dx.doi.org/10.1007/978-3-642-25591-5_39

[22] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, International Journal of High Performance Computing Applications 19 (2005) 49–66. doi:10.1177/1094342005051521.
URL http://dx.doi.org/10.1177/1094342005051521

[23] P. Patarasuk, X. Yuan, Bandwidth efficient allreduce operation on tree topologies, in: IEEE IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments, 2007. doi:10.1109/IPDPS.2007.370405.
URL http://dx.doi.org/10.1109/IPDPS.2007.370405

[24] S. Smolyak, Quadrature and interpolation formulas for tensor products of certain classes of functions, Soviet Mathematics, Doklady 4 (1963) 240–243.

[25] D. Pflüger, Spatially adaptive refinement, in: J. Garcke, M. Griebel (Eds.), Sparse Grids and Applications, LNCSE, Springer, Berlin Heidelberg, 2012, pp. 243–262.

[26] M. Hegland, Additive sparse grid fitting, in: Proceedings of the Fifth International Conference on Curves and Surfaces, Saint-Malo, France 2002, Nashboro Press, 2003, pp. 209–218.

[27] M. Hegland, J. Garcke, V. Challis, The combination technique and some generalisations, Linear Algebra and its Applications 420 (2–3) (2007) 249–275. doi:doi:10.1016/j.laa.2006.07.014.

[28] C. Kowitz, M. Hegland, The sparse grid combination technique for computing eigenvalues in linear gyrokinetics, Procedia Computer Science 18 (0) (2013) 449 – 458, 2013 International Conference on Computational Science. doi:http://dx.doi.org/10.1016/j.procs.2013.05.208.
URL http://www.sciencedirect.com/science/article/pii/S1877050913003517
[29] M. Holtz, Sparse Grid Quadrature in High Dimensions with Applications in Finance and Insurance., Vol. 77 of Lecture Notes in Computational Science and Engineering, Springer, 2011.
[30] H.-J. Bungartz, A. Heinecke, D. Pflüger, S. Schraufstetter, Option pricing with a direct adaptive sparse grid approach, Journal of Computational and Applied Mathematics 236 (15) (2011) 3741 —- 3750, online Okt. 2011.
[31] D. Butnaru, D. Pflüger, H.-J. Bungartz, Towards high-dimensional computational steering of precomputed simulation data using sparse grids, in: Proceedings of the International Conference on Computational Science (ICCS) 2011, Vol. 4 of Procedia CS, Tsukaba, Japan, Springer-Verlag, 2011, pp. 56–65.
[32] G. Buse, R. Jacob, D. Pflüger, A. Murarasu, A non-static data layout enhancing parallelism and vectorization in sparse grid algorithms, in: ISPDC 2012, IEEE, Munich, 2012.
[33] D. Pflüger, Spatially Adaptive Sparse Grids for High-Dimensional Problems, Verlag Dr. Hut, München, 2010.
URL http://www5.in.tum.de/pub/pflueger10spatially.pdf
[34] J. Garcke, M. Griebel, On the parallelization of the sparse grid approach for data mining, in: S. Margenov, J. Waśniewski, P. Yalamov (Eds.), Large-Scale Scientific Computing, Vol. 2179 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 22–32.
[35] H.-J. Bungartz, D. Pflüger, S. Zimmer, Adaptive sparse grid techniques for data mining, in: H. Bock, E. Kostina, X. Hoang, R. Rannacher (Eds.), Modelling, Simulation and Optimization of Complex Processes 2006, Proc. Int. Conf. HPSC, Hanoi, Vietnam, Springer-Verlag, 2008, pp. 121–130.
URL http://www5.in.tum.de/pub/pflueger06adaptive.pdf
[36] M. Griebel, The combination technique for the sparse grid solution of pde's on multiprocessor machines, in: Parallel Processing Letters, 1992, pp. 61–70.
[37] M. Griebel, W. Huber, U. Rüde, T. Störtkuhl, The combination technique for parallel sparse-grid-preconditioning or -solution of pde's on workstation networks, in: L. Bougé, M. Cosnard, Y. Robert, D. Trystram (Eds.), Parallel Processing: CONPAR 92—VAPP V, Vol. 634 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1992, pp. 217–228.
[38] J. Garcke, M. Griebel, On the parallelization of the sparse grid approach for data mining, in: Proceedings of the Third International Conference on Large-Scale Scientific Computing-Revised Papers, LSSC '01, Springer-Verlag, London, UK, UK, 2001, pp. 22–32.
URL http://dl.acm.org/citation.cfm?id=645740.666764
[39] J. Garcke, M. Hegland, O. Nielsen, Parallelisation of sparse grids for large scale data analysis, in: Proceedings of the 2003 International Conference on Computational Science: PartIII, ICCS'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 683–692.
URL http://dl.acm.org/citation.cfm?id=1762418.1762493
[40] J. Benk, D. Pflüger, Hybrid parallel solutions of the black-scholes pde with the truncated combination technique, in: Proceedings of the HPCS conference, Madrid, 2012.
[41] M. Heene, C. Kowitz, D. Pflüger, Load balancing for massively parallel computations with the sparse grid combination technique, in: Parallel Computing - Accelerating Computational Science and Engineering (CSE), Vol. 25 of Advances in Parallel Computing, IOS Press, 2014.
[42] K.-H. Huang, J. A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Trans. Comput. 33 (6) (1984) 518–528. doi:10.1109/TC.1984.1676475.
URL http://dx.doi.org/10.1109/TC.1984.1676475
[43] B. Harding, M. Hegland, A robust combination technique, in: CTAC-2012, Vol. 54 of ANZIAM J., 2013, pp. C394–C411.
[44] J. Larson, M. Hegland, B. Harding, S. Roberts, L. Stals, A. Rendell, P. Strazdins, M. Ali, C. Kowitz, R. Nobes, J. Southern, N. Wilson, M. Li, Y. Oishi, Fault-tolerant grid-based solvers: Combining concepts from sparse grids and mapreduce, in: ICCS 2014, Vol. 18 of Procedia Computer Science, Elsevier, 2013, pp. 130–139. doi:http://dx.doi.org/10.1016/j.procs.2013.05.176.
URL http://www.sciencedirect.com/science/article/pii/S1877050913003190
[45] B. Harding, M. Hegland, A parallel fault tolerant combination technique, in: Parallel Computing - Accelerating Computational Science and Engineering (CSE), Vol. 25 of Advances in Parallel Computing, IOS Press, 2014.
[46] M. Griebel, W. Huber, T. Störtkuhl, C. Zenger, On the parallel solution of 3d PDEs on a network of workstations and on vector computers, in: Parallel Computer Architectures: Theory, Hardware, Software, Applications, LNCS, Springer-Verlag, London, UK, UK, 1993, pp. 276–291. doi:10.1007/3-540-57307-0_41.
URL http://dl.acm.org/citation.cfm?id=647824.736562
[47] C. Kranz, Untersuchungen zur kombinationstechnik bei der numerischen strömungssimulation auf versetzten gittern, Dissertation, TU München (2002).
URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/kranz.pdf
[48] A. Murarasu, J. Weidendorfer, G. Buse, D. Butnaru, D. Pflüger, Compact data structure and scalable algorithms for the sparse grid technique, in: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, ACM, New York, NY, USA, 2011, pp. 25–34. doi:10.1145/1941553.1941559.
URL http://doi.acm.org/10.1145/1941553.1941559
[49] A. F. Murarasu, G. Buse, D. Pflüger, J. Weidendorfer, A. Bode, fastsg: A fast routines library for sparse grids, Procedia CS 9 (2012) 354–363.
[50] R. Jacob, Efficient regular sparse grid hierarchization by a dynamic memory layout, in: J. Garcke, D. Pflüger (Eds.), Sparse Grids and Applications - Munich 2012, Vol. 97 of Lecture Notes in Computational Science and Engineering, Springer International Publishing, 2014, pp. 195–219. doi:10.1007/978-3-319-04537-5_8.
URL http://dx.doi.org/10.1007/978-3-319-04537-5_8
[51] P. Hupp, Performance of unidirectional hierarchization for component grids virtually maximized, in: 2014 International Conference on Computational Science, Vol. 29 of Procedia Computer Science, Elsevier, 2014, pp. 2272–2283. doi:http://dx.doi.org/10.1016/j.procs.2014.05.212.
URL http://www.sciencedirect.com/science/article/pii/S1877050914003895
[52] P. Hupp, R. Jacob, Cache-optimal component grid hierarchization outperforming the unidirectional algorithm, in preparation.

[53] H.-J. Bungartz, M. Griebel, Sparse grids, Acta Numerica 13 (2004) 147–269. doi:10.1017/S0962492904000182.
[54] S. Andersson, Best practices – Hermit (talk), PRACE Spring School 2012/05/16 (2012).
URL http://www.training.prace-ri.eu/uploads/tx_pracetmo/BestPracticeHermit.pdf
[55] K. W. Schulz, Experiences from the deployment of TACC's stampede system (talk), lugano Switzerland (March 2013).
URL http://www.hpcadvisorycouncil.com/events/2013/Switzerland-Workshop/Presentations/Day_1/7_TACC.pdf
[56] S. Kumar, Challenges of exascale messaging library design: Case study with blue gene machines (talk), cASS Workshop, IPDPS (May 2012).
URL http://www.ccs3.lanl.gov/cass2012/talks/kumar.pdf