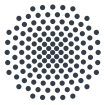

Universität Stuttgart

A massively parallel combination technique for the solution of high-dimensional PDEs

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart sowie dem SRC SimTech zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Mario Heene
aus Nürnberg

Hauptberichter:	Jun.-Prof. Dr. Dirk Pflüger
Mitberichter:	Prof. Dr. Markus Hegland Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch

Tag der mündlichen Prüfung: 04.10.2017

Institut für Parallele und Verteilte Systeme der Universität Stuttgart

2018

Abstract

The solution of high-dimensional problems, especially high-dimensional partial differential equations (PDEs) that require the joint discretization of more than the usual three spatial dimensions and time, is one of the grand challenges in high performance computing (HPC). Due to the exponential growth of the number of unknowns – the so-called curse of dimensionality, it is in many cases not feasible to resolve the simulation domain as fine as required by the physical problem. Although the upcoming generation of exascale HPC systems theoretically provides the computational power to handle simulations that are out of reach today, it is expected that this is only achievable with new numerical algorithms that are able to efficiently exploit the massive parallelism of these systems.

The sparse grid combination technique is a numerical scheme where the problem (e.g., a high-dimensional PDE) is solved on different coarse and anisotropic computational grids (so-called component grids), which are then combined to approximate the solution with a much higher target resolution than any of the individual component grids. This way, the total number of unknowns being computed is drastically reduced compared to the case when the problem is directly solved on a regular grid with the target resolution. Thus, the curse of dimensionality is mitigated.

The combination technique is a promising approach to solve high-dimensional problems on future exascale systems. It offers two levels of parallelism: the component grids can be computed in parallel, independently and asynchronously of each other; and the computation of each component grid can be parallelized as well. This reduces the demand for global communication and synchronization, which is expected to be one of the limiting factors for classical discretization techniques to achieve scalability on exascale systems. Furthermore, the combination technique enables novel approaches to deal with the increasing fault rates expected from these systems. With the fault-tolerant combination technique it is possible to recover from failures without time-consuming checkpoint-restart mechanisms.

In this work, new algorithms and data structures are presented that enable a massively parallel and fault-tolerant combination technique for time-dependent PDEs on large-scale HPC systems. The scalability of these algorithms is demonstrated on up to 180 225 processor cores on the supercomputer *Hazel Hen*. Furthermore, the parallel combination technique is applied to gyrokinetic simulations in GENE, a software for the simulation of plasma microturbulence in fusion devices.

Zusammenfassung

Die Lösung von hochdimensionalen Problemen, insbesondere hochdimensionalen partiellen Differentialgleichungen (PDGL), die die Diskretisierung der üblichen drei Raumdimensionen und zusätzlich der Zeit erfordern, ist eine der großen Herausforderungen im High Performance Computing (HPC). Aufgrund des exponentiellen Anstiegs der Anzahl an Unbekannten, dem so genannten Fluch der Dimensionalität, ist es in vielen Fällen nicht machbar das Simulationsgebiet so fein aufzulösen, wie es für das physikalische Problem erforderlich wäre. Die bevorstehende Generation von Exascale-HPC-Systemen stellt zwar theoretisch die Rechenleistung bereit, um Simulationen zu behandeln die heutzutage außer Reichweite sind. Es wird jedoch erwartet, dass dies nur mit neuen Algorithmen möglich ist, die die extreme Parallelität dieser Systeme effizient ausnutzen können.

Die Dünngitter-Kombinationstechnik ist ein numerisches Schema, bei dem das Problem (z.B. eine hochdimensionale PDGL) auf verschiedenen groben und anisotropen Rechengittern (sogenannten Komponentengittern) gelöst wird, die dann kombiniert werden, um die Lösung mit einer wesentlich höheren Zielauflösung zu approximieren. Auf diese Weise wird die Anzahl an Unbekannten im Vergleich zur direkten Lösung des Problems auf einem regulären Gitter, das der Zielauflösung entspricht, drastisch verringert.

Die Kombinationstechnik ist ein vielversprechendes numerisches Verfahren um hochdimensionale Probleme auf zukünftigen Exascale-Systemen zu lösen. Sie weist zwei Parallelisierungsebenen auf: die Komponentengitter können parallel, unabhängig und asynchron von einander berechnet werden; und die Berechnung jedes Komponentengitters kann ebenfalls parallelisiert werden. Das reduziert den Bedarf an globaler Kommunikation und Synchronisierung, von dem man erwartet, dass es einer der limitierenden Faktoren ist, um Skalierbarkeit auf Exascale-Systemen zu erreichen. Außerdem ermöglicht die Kombinationstechnik neuartige Verfahren, um die für diese Systeme erwarteten steigenden Fehlerraten in den Griff zu bekommen. Mit der fehlertoleranten Kombinationstechnik ist die Fehlerbehandlung ohne zeitaufwendige Checkpoint-Restart Verfahren möglich.

In dieser Arbeit werden neue Algorithmen und Datenstrukturen vorgestellt, die eine hochparallele und fehlertolerante Kombinationstechnik für zeitabhängige PDGL auf großen HPC-Systemen ermöglichen. Die Skalierbarkeit dieser Algorithmen wird auf 180 225 Prozessorkernen des Supercomputers Hazel Hen demonstriert. Außerdem wird die parallele Kombinationstechnik auf gyrokinetische Simulationen mit GENE angewendet, einer Software zur Simulation von Plasmamikroturbulenz in Fusionsreaktoren.

Contents

1. Introduction **2**

2. Sparse grids and the combination technique **5**
2.1. Function interpolation 5
2.2. Hierarchical basis 7
2.3. Hierarchization and dehierarchization 9
2.4. Sparse grids 11
2.5. The sparse grid combination technique 12
2.6. Time integration 15
2.7. Investigation of different combination strategies 17
2.7.1. Costs of different combination strategies 18
2.7.2. Optimal communication schemes for the combination in the hierarchical basis 19

3. Development of a massively parallel combination technique **28**
3.1. Parallelization concept 28
3.2. Load balancing 33
3.2.1. Cost models 34
3.2.2. Accuracy of the *anisotropic cost model* 37
3.2.3. Scheduling experiments 40
3.2.4. Rebalancing after the combination step 42
3.3. Efficient and scalable combination 44
3.3.1. Distributed hierarchization and dehierarchization 46
3.3.2. Scalability of the distributed hierarchization 50
3.3.3. Local reduction / scatter of component grids inside the process group 56
3.3.4. Global reduction of the combined solution 63
3.3.5. Experimental results 65
3.4. Evaluation of the combined solution for post-processing 68
3.5. Application interface 70
3.6. Related work 71

4. A scalable fault-tolerant combination technique **73**
4.1. Fault-tolerant combination technique 75
4.2. Implementation 78
4.3. Fault simulation software layer 80
4.4. Experiments and results 83
4.4.1. Convergence 84

4.4.2. Scalability 85

5. The combination technique applied to gyrokinetic simulations with GENE 88
5.1. GENE 89
5.1.1. Interface 91
5.1.2. Alignment of grids and boundary conditions 92
5.2. Experiments and results 97
5.2.1. Eigenvalue λ 99
5.2.2. Eigenvector g 102
5.2.3. Other quantities of interest in GENE 115
5.3. Outlook 119

6. Conclusions 120

A. HPC systems used for the experiments 122

B. GENE parameter files used in the experiments 124
B.1. Linear local simulation 124
B.2. Non-linear global simulation 125

1. Introduction

One of the grand challenges in high-performance computing (HPC) is the solution of high-dimensional problems, such as high-dimensional partial differential equations (PDEs), which require the joint discretization of more than the usual three spatial dimensions and time. Due to the exponential growth of the number of unknowns with the number of dimensions (the so-called curse of dimensionality), resolving the simulation domain as fine as it would be required by the physical problem is prohibitively expensive in many cases. One problem where this can be observed is the simulation of plasma microturbulence in a fusion device. In order to perform physically accurate turbulence simulations with the underlying five-dimensional gyrokinetic equations, the resolutions required for a large fusion device, such as the world's flagship fusion experiment ITER, quickly exceed the computational capabilities even of the largest HPC systems that are available today.

Although the upcoming *exascale* generation of HPC systems (systems that offer a compute performance of more than 10^{18} floating point operations per second) theoretically provides the computational power to handle simulations that are out of scope today, it is expected that this is only achievable with new numerical algorithms that are able to efficiently exploit the massive parallelism of these systems, which will consist of millions of processing units [61, 24]. One of the major issues that limit the scalability of parallel scientific applications already on today's *petascale* systems are the global communication and synchronization of the individual processes. This problem will be amplified even more by the significantly increased parallelism of future systems and thus requires new approaches to reduce the communication/synchronization bottleneck. Another key issue of exascale systems is that, with growing size and complexity of these systems, also the rates at which failures occur are predicted to increase significantly [18, 19]. Therefore, new techniques are necessary that can detect and correct faults in an efficient way, without expensive checkpoint-restart mechanisms.

The sparse grid combination technique is a numerical scheme where the problem (e.g., a high-dimensional PDE) is solved on different coarse and anisotropic computational grids (so-called component grids), which are then combined to approximate the solution on a grid with a much higher target resolution. The combination technique mitigates the curse of dimensionality, and thus a similar accuracy can be achieved with significantly lower computational resources than it would be possible when directly solving the problem with the target resolution. This allows us to compute target resolutions that would otherwise be out of scope due to the exploding number of grid points.

The combination technique is a promising numerical approach to solve high-dimensional problems on future exascale systems. It offers two levels of parallelism: the component grids can be computed in parallel, independently and asynchronously of each other; furthermore, the computation of each component grid can be parallelized as well. This

reduces the demand for global communication and synchronization, which is expected to be one of the limiting factors with classical discretization techniques to achieve scalability on future exascale systems. Furthermore, the combination technique enables novel approaches to deal with the increasing fault rates expected from these systems. With the *fault-tolerant combination technique* it is possible to recover from failures without time-consuming checkpoint-restart mechanisms.

This work was part of the project *EXAHD* [75] of the priority program *Software for Exascale Computing (SPPEXA)* [2, 17] of the *German Research Foundation (DFG)*. The major goal of EXAHD is to push the computational limits of high-dimensional problems on massively parallel systems with the combination technique and to demonstrate this at the example of plasma turbulence simulations with the simulation code GENE. The main contribution of this work to the project was the development of new algorithms and data structures that enable a massively parallel and fault-tolerant combination technique for time-dependent PDEs on HPC systems. Furthermore, results are presented for the parallel combination technique applied to initial-value computations in GENE.

The outline of this work is as follows. Chapter 2 provides an introduction to sparse grids and the combination technique. Furthermore, different approaches to combine the component grids as efficient as possible are investigated from a theoretical point of view. Chapter 3 is the core of this work. Here, the key components for the implementation of a massively parallel combination technique are discussed, including a scalable parallelization concept, effective load balancing schemes and efficient and scalable algorithms and data structures for the combination step. In Chapter 4 a fault-tolerant version of the parallel combination technique is presented and applied to an advection-diffusion equation implemented in the PDE-framework DUNE [14]. Additionally, in order to be able to perform experiments on HPC systems where a fault-tolerant MPI implementation is not yet available, a *fault simulation software layer* is introduced. In Chapter 5 the parallel combination technique is applied to initial-value simulations in GENE in order to demonstrate that the combination technique can be successfully applied to this problem.

Parts of this work have already been published in:

[59] Philipp Hupp, Riko Jacob, Mario Heene, Dirk Pflüger, and Markus Hegland. Global communication schemes for the sparse grid combination technique. (Section 2.7)

[58] Philipp Hupp, Mario Heene, Riko Jacob, and Dirk Pflüger. Global communication schemes for the numerical solution of high-dimensional PDEs. (Section 2.7)

[75] Dirk Pflüger, Hans-Joachim Bungartz, Michael Griebel, Frank Jenko, Tilman Dannert, Mario Heene, Alfredo Parra Hinojosa, Christoph Kowitz, and Peter Zaspel. EXAHD: An Exa-Scalable Two-Level Sparse Grid Approach for Higher-Dimensional Problems in Plasma Physics and Beyond. (Sections 2.7 and 3.2)

[49] Mario Heene, Christoph Kowitz, and Dirk Pflüger. Load balancing for massively parallel computations with the sparse grid combination technique. (Section 3.2)

[50] Mario Heene and Dirk Pflüger. Efficient and scalable distributed-memory hierarchization algorithms for the sparse grid combination technique. (Section 3.3)

[51] Mario Heene and Dirk Pflüger. Scalable algorithms for the solution of higher-dimensional PDEs. (Section 3.3)

[48] Mario Heene, Alfredo Parra Hinojosa, Hans-Joachim Bungartz, and Dirk Pflüger. A massively-parallel, fault-tolerant solver for high-dimensional PDEs. (Chapter 4)

[55] Alfredo Parra Hinojosa, Christoph Kowitz, Mario Heene, Dirk Pflüger, and Hans-Joachim Bungartz. Towards a fault-tolerant, scalable implementation of GENE. (Chapters 4 and 5)

2. Sparse grids and the combination technique

Sparse grids are a method to mitigate the curse of dimensionality for high-dimensional problems. For certain classes of functions, this method allows to reduce the number of grid points significantly compared to the full grid, without a significant deterioration of the convergence rate. The term *Sparse Grids* was first introduced by Zenger [86] as a method for solving PDEs with a hierarchical basis. A comprehensive review of sparse grids can be found in [16].

The *Sparse Grid Combination Technique* was introduced by Griebel et al. [39]. A similar approach was introduced by Smolyak [78] already in 1963 for quadrature and interpolation. The idea of the combination technique is to obtain a sparse grid approximation by a superposition of component solutions computed on coarse and anisotropic regular grids. One advantage of the combination technique is that it is not necessary to use a hierarchical basis. The component solutions can be computed with a regular nodal basis and thus the combination technique can be implemented with much simpler data structures and algorithms than sparse grids. Furthermore, the combination technique is inherently parallel, which makes it a good candidate for the application to the large-scale and massively parallel applications targeted in this work.

In the following, the theoretical background of sparse grids and the combination technique will be introduced briefly. Furthermore, the special requirements for the application of the combination technique to large-scale time-dependent PDEs will be discussed.

2.1. Function interpolation

The goal of function interpolation is to approximate a d-dimensional function $f(\vec{x}) : \Omega \to \mathbb{R}$, with $\Omega \subseteq \mathbb{R}^d$, by a linear combination of basis functions $\varphi_i(\vec{x}) : \Omega \to \mathbb{R}$

$$f(\vec{x}) \approx \sum_i \alpha_i \varphi_i(\vec{x}). \tag{2.1}$$

In contrast to plain approximation, for interpolation the value of f is known at a given set of grid points $\vec{x}_i$. There are various possibilities to choose the basis functions. In this work a piecewise linear basis is used. This is a widely used basis, especially in the context of *Finite Element Methods*.

The problem domain $\Omega := [0, 1]^d$ is discretized with a regular grid $\Omega_{\vec{n}}$. The vector $\vec{n} \in \mathbb{N}^d$ denotes the discretization level n_k and the corresponding mesh width $h_k := 2^{-n_k}$

in each dimension k. Thus, the resulting grid points $\vec{x}_i$ have equidistant spacing within each coordinate direction of Ω.

For a one-dimensional grid the basis function is the so-called *hat function*

$$\varphi_i(x) = \max\left(1 - \frac{|x - x_i|}{h}, 0\right). \tag{2.2}$$

The hat function is chosen so that it is one at the grid point itself, zero at the two neighboring grid points and linear in-between. Furthermore, it has local support, i.e., it is non-zero only in the interval $]x_i - h, x_i + h[$. The hat function can be extended to multiple dimensions by a tensor product of one-dimensional hat functions

$$\varphi_i(\vec{x}) = \prod_{k=1}^{d} \max\left(1 - \frac{|(\vec{x})_k - (\vec{x}_i)_k|}{h_k}, 0\right). \tag{2.3}$$

For each grid point $\vec{x}_i$ in $\Omega_{\vec{n}}$ there is exactly one corresponding basis function φ_i. The number of grid points in $\Omega_{\vec{n}}$ is

$$N = |\Omega_{\vec{n}}| = \prod_{k=1}^{d} (2^{l_k} \pm 1) \tag{2.4}$$

($+1$ if the grid has boundary points in dimension k and -1 if not). The function space $V_{\vec{n}}$ spanned by these basis functions that corresponds to $\Omega_{\vec{n}}$ is defined as

$$V_{\vec{n}} := \operatorname{span}\{\varphi_i : 0 \leq i \leq N - 1\}. \tag{2.5}$$

The corresponding interpolant $u_{\vec{n}} \in V_{\vec{n}}$ is defined as

$$u_{\vec{n}}(\vec{x}) := \sum_{i=0}^{N-1} \alpha_i \varphi_i(\vec{x}). \tag{2.6}$$

In order to approximate the function f with this interpolant one would choose $\alpha_i = f(\vec{x}_i)$ and thus

$$f(\vec{x}) \approx \sum_{i=0}^{N-1} f(\vec{x}_i) \varphi_i(\vec{x}). \tag{2.7}$$

In the following, $V_{\vec{n}}$ is called space of *nodal basis functions* and the corresponding interpolant $u_{\vec{n}}$ is called *nodal basis representation*. The grid $\Omega_{\vec{n}}$ is called *full grid*.

It can be shown that for a sufficiently smooth function f the approximation error $\|f - u_{\vec{n}}\|_2$ of the interpolant $u_{\vec{n}}$ with respect to the L_2-norm decreases asymptotically with $\mathcal{O}(h^2)$. The L_2-norm is defined as

$$\|f\|_2 := \left(\int_\Omega |f(\vec{x})|^2 dx\right)^{1/2}. \tag{2.8}$$

The computational costs, in terms of function evaluations or number of grid points, are of $\mathcal{O}(2^{nd})$ – they increase exponentially with the dimensionality. Note that these asymptotes are formulated for isotropic discretizations, i.e., $n_k = n \; \forall k$ and $h_k = 2^{-n}$.

2.2. Hierarchical basis

In order to represent a function on a sparse grid, a *hierarchical basis* is used. The hierarchical basis decomposes the nodal space into a set of hierarchical subspaces $W_{\vec{l}}$,

$$V_{\vec{n}} = \bigoplus_{\vec{l} \leq \vec{n}} W_{\vec{l}}. \tag{2.9}$$

The so-called *level vector* $\vec{l} \in \mathbb{N}^d$ denotes the discretization level in each dimension of $W_{\vec{l}}$. In the following, for two arbitrary vectors $\vec{a}, \vec{b} \in \mathbb{N}^d$ we define

$$\vec{a} \leq \vec{b} \iff a_k \leq b_k \quad \text{and} \quad \vec{a} < \vec{b} \iff a_k < b_k. \tag{2.10}$$

Each subspace $W_{\vec{l}}$ is spanned by a set of multi-dimensional hat functions

$$\phi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{k=1}^{d} \max\left(1 - \frac{\vec{x}_k - (\vec{x}_{\vec{l},\vec{i}})_k}{2^{-l_k}}, 0\right), \tag{2.11}$$

which are centered at the corresponding grid points $\vec{x}_{\vec{l},\vec{i}}$. The index vector $\vec{i} \in \mathbb{N}^d$ denotes the index of the grid point $\vec{x}_{\vec{l},\vec{i}}$ in each dimension k, where

$$1 \leq i_k \leq 2^{l_k} - 1, \ i \text{ odd}. \tag{2.12}$$

Note that this notation does not consider grid points at the domain's boundary. The treatment of boundary points is addressed later. The set of index vectors corresponding to $W_{\vec{l}}$ is denoted $I_{\vec{l}}$. This results in the corresponding coordinates

$$(\vec{x}_{\vec{l},\vec{i}})_k := 2^{-l_k} i_k. \tag{2.13}$$

The hierarchical basis is created by the same standard hat functions as the nodal basis. Note that the supports of the basis functions within a hierarchical subspace do not overlap. Furthermore, all basis functions of a hierarchical subspace $W_{\vec{l}}$, with $\vec{l}' > \vec{l}$, are zero at the grid points of $W_{\vec{l}}$.

In the so-called *hierarchical basis representation* the interpolant $u_{\vec{n}} \in V_{\vec{n}}$ is written as

$$u_{\vec{n}}(\vec{x}) = \sum_{\vec{l} \leq \vec{n}} \sum_{\vec{i} \in I_{\vec{l}}} \omega_{\vec{l},\vec{i}} \phi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.14}$$

Note that in general the *hierarchical coefficients* $\omega_{\vec{l},\vec{i}}$ are not the function values at the grid points, as in Equation (2.7). They are obtained by *hierarchization* as described in the next section.

Figure 2.1 compares the nodal basis representation and the hierarchical basis representation for the approximation of a one-dimensional function without points at the boundary. Here, one can easily see the equivalency of the interpolant for the two different bases.

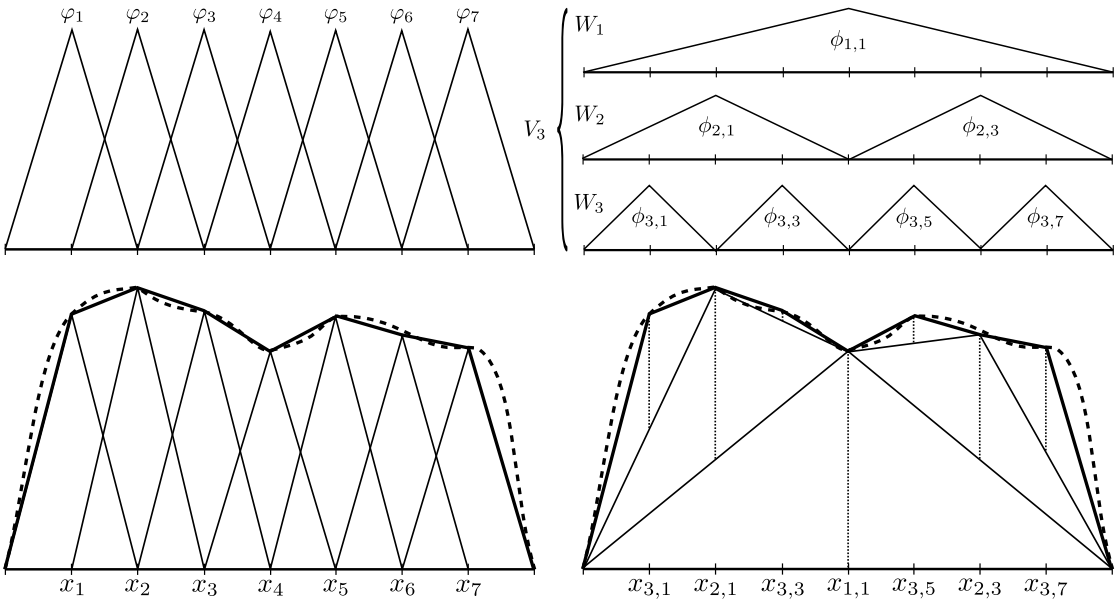

Figure 2.1.: Interpolation with the nodal basis representation (**left**) and the hierarchical basis representation (**right**).

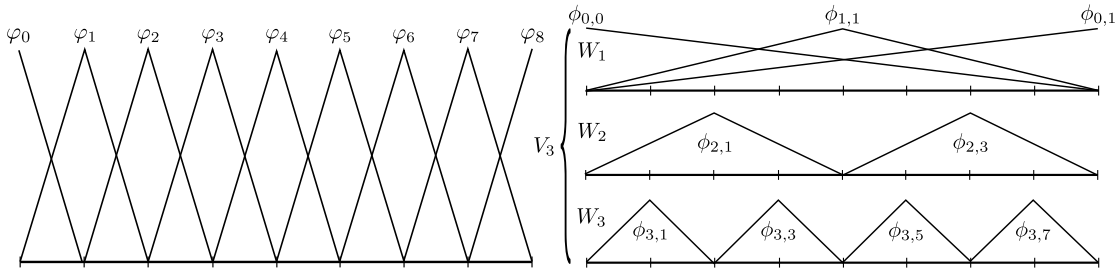

Figure 2.2.: Nodal basis (**left**) and hierarchical basis (**right**) with points on the domain boundary.

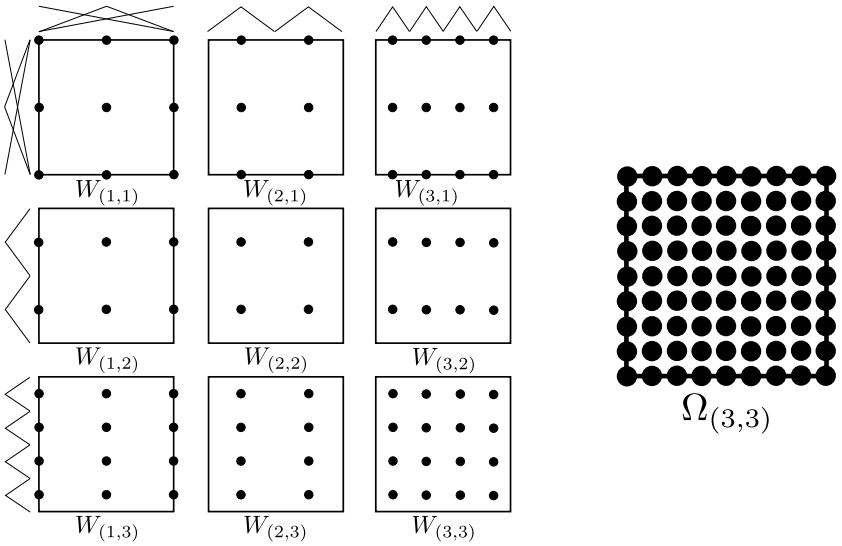

Figure 2.3.: Hierarchical subspaces of the full grid $\Omega_{(3,3)}$.

For discretizations having grid points on the domain's boundary $\partial\Omega$, there are different variants of the hierarchical basis (see [73]). In this work the variant is used which extends the coarsest hierarchical W_1 subspace by two additional basis functions of level 0, $\phi_{0,0}$ and $\phi_{0,1}$. This variant of a hierarchical basis is a natural choice for the combination technique, because it creates a basis which is equivalent to the nodal basis representation. For the nodal basis it is common to simply use the same basis functions for the grid points on the boundary as for the inner grid points. This is visualized in Figure 2.2. Figure 2.3 shows the hierarchical subspaces of a two-dimensional full grid with points on the boundary.

2.3. Hierarchization and dehierarchization

The operation to transform a function $u_{\vec{n}} \in V_{\vec{n}}$ from the nodal basis representation into the hierarchical basis representation is called *hierarchization*. Correspondingly, the inverse operation is called *dehierarchization*.

For a one-dimensional function the hierarchical coefficient ω_i corresponding to the grid point x_i is obtained by

$$\omega_i = f_i - \frac{1}{2}\left(f_{i,\text{left}} + f_{i,\text{right}}\right). \tag{2.15}$$

In accordance with Equation (2.7), it holds $f_i := f(x_i)$. $f_{i,\text{left}}$ and $f_{i,\text{right}}$ are the function values corresponding to the left and right *hierarchical predecessor*, $x_{i,\text{left}}$ and $x_{i,\text{right}}$, of the grid point x_i. The hierarchical predecessors are the grid points closest to x_i that are in a hierarchical subspace W_l with a lower discretization level than the subspace which

contains x_i. The indices of the hierarchical predecessors are computed as

$$i, \text{left} := i - 2^{n-l_i},$$
$$i, \text{right} := i + 2^{n-l_i},$$

where l_i denotes the discretization level of the hierarchical subspace that contains x_i and n is the discretization level of V_n. Usually, one iterates over the grid points in a level-wise fashion, starting on the highest level. This means one iterates over all grid points of a particular hierarchical subspace, before one continues with the next subspace. This order has the advantage that the hierarchization can be done in-place, i.e., without an additional copy of the function values, which is desirable in some situations.

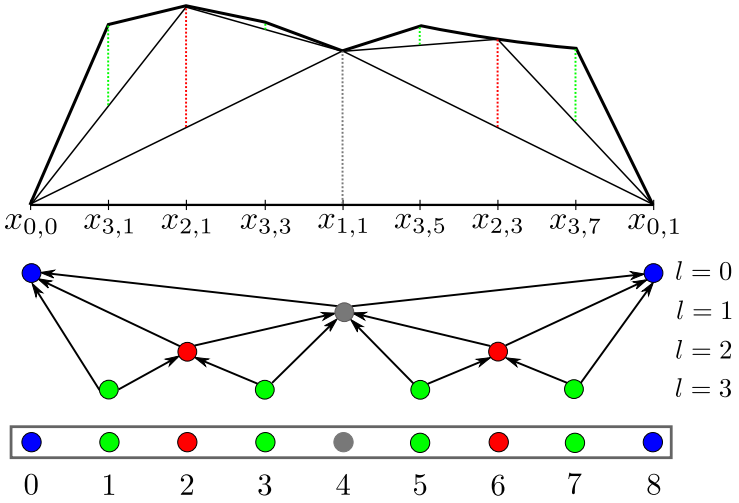

Figure 2.4.: Hierarchization/dehierarchization of a one-dimensional function. The arrows indicate the left and right hierarchical predecessor of each grid point.

The rule for the dehierarchization can be obtained by solving Equation (2.15) for f_i

$$f_i = \omega_i + \frac{1}{2}\left(f_{i,\text{left}} + f_{i,\text{right}}\right). \tag{2.16}$$

The formula indicates the inherent order for the dehierarchization: before a grid point x_i can be dehierarchized, its hierarchical predecessors must have been dehierarchized already. This is fulfilled when one iterates over the grid points in a level-wise fashion, starting on the lowest level. Figure 2.4 visualizes the data dependencies for the in-place hierarchization and dehierarchization.

The so-called *unidirectional principle* extends this procedure to multiple dimensions. Due to the product structure of the basis functions, it is possible to hierarchize (or dehierarchize) a multi-dimensional function by a series of one-dimensional hierarchization (dehierarchization) operations. Unidirectional hierarchization and dehierarchization are discussed in detail in Section 3.3.1.

2.4. Sparse grids

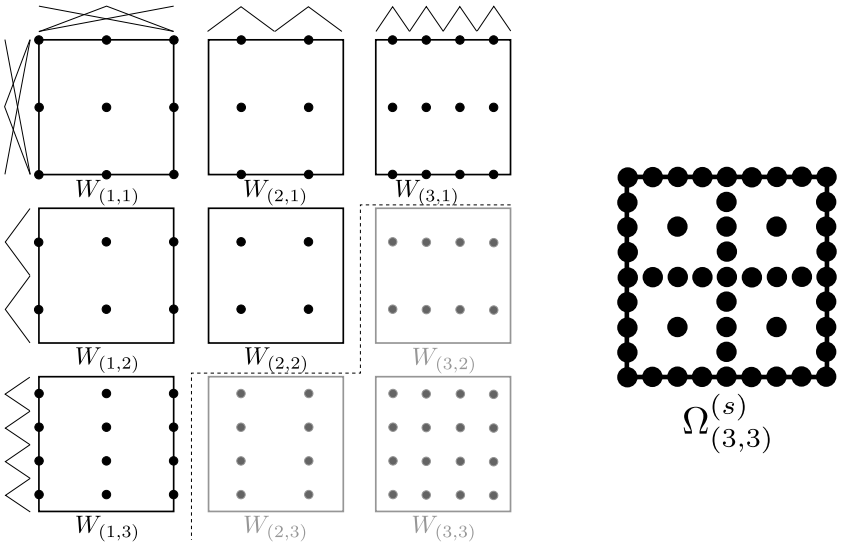

Figure 2.5.: Hierarchical subspaces of the sparse grid $\Omega^{(s)}_{(3,3)}$. The grey subspaces have been removed from the full grid $\Omega_{(3,3)}$.

If the function that is approximated is sufficiently smooth, it can be observed that the hierarchical coefficients decrease with increasing level of the hierarchical subspaces. This can, for example, be observed in Figure 2.1. This means, the higher the discretization level of a hierarchical subspace, the lesser it contributes to the approximation. The idea of *sparse grids* is to omit the subspaces with the highest level sums in order to reduce the number of grid points.

It can be shown [16] that for a function f with bounded second order mixed derivatives

$$\mathcal{D}^2 f = \frac{\partial 2^d}{\partial x_1^2 \dots \partial x_d^2} \tag{2.17}$$

the optimal choice of hierarchical subspaces is

$$V_n^{(s)} := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}, \tag{2.18}$$

where

$$|\vec{l}|_1 := \sum_{k=1}^{d} l_k \tag{2.19}$$

denotes the level sum. This choice of subspaces optimizes the approximation error $f - u^{(s)}_{\vec{n}}$ of the sparse grid interpolant $u^{(s)}_{\vec{n}}$ with respect to the L_2-norm (see Equation (2.8)) or the maximum norm

$$\|f\|_\infty := \max_{\vec{x} \in \Omega} |f(\vec{x})|. \tag{2.20}$$

Figure 2.5 shows the set of subspaces according to Equation (2.18) for a two-dimensional sparse grid of level 3 with boundary points. Compared to the corresponding full grid $\Omega_{(3,3)}$, the grey subspaces have been removed.

The number of grid points, and thus the number of function evaluations to determine the interpolant, of sparse grids are of $\mathcal{O}(2^n \log(2^n)^{d-1})$. Thus the number of grid points grows significantly slower than for full grids with $\mathcal{O}(2^{nd})$. However, the asymptotic decrease of the approximation error is in $\mathcal{O}(h^2 \log(h^{-1})^{d-1})$ and thus only slightly worse than for full grids.

Note that Equation (2.18) is only formulated for isotropic discretizations, i.e., $n = n_k$ for all entries n_k of $\vec{n}$. However, the set of hierarchical subspaces can easily be adapted to anisotropic discretizations, as well. In fact, with sparse grids full spatial adaptivity is possible (see, e.g., [73]), which is one of the most desirable features of this method.

2.5. The sparse grid combination technique

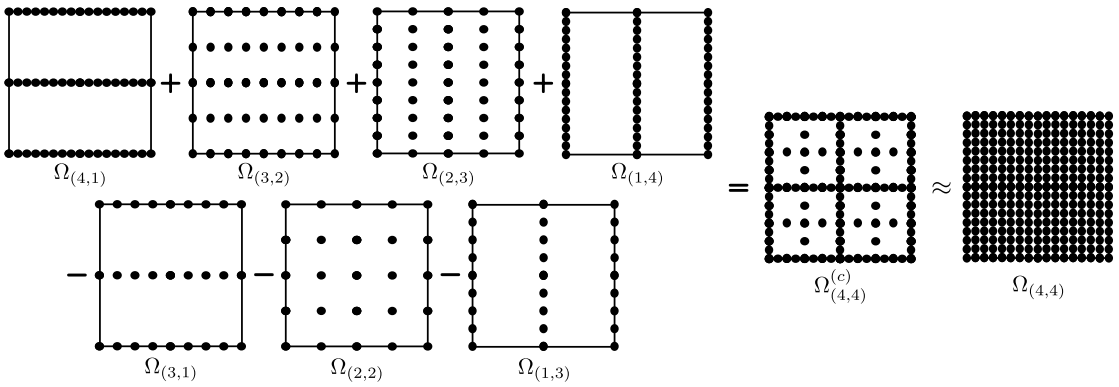

Figure 2.6.: The classical combination technique in two dimensions with $n = 4$. Seven component grids are combined to obtain a sparse grid approximation (on the grid $\Omega^{(s)}_{(4,4)}$) to the full grid solution on the grid $\Omega_{(4,4)}$.

The idea behind the *sparse grid combination technique* [36] is to approximate a function f on a sparse grid by a linear combination of component solutions $u_{\vec{l}}$ on regular grids. Each $u_{\vec{l}}$ is an approximation of f that has been computed on a coarse and anisotropic, but regular component grid $\Omega_{\vec{l}}$.

The *classical* combination technique [36] was formulated as

$$f \approx u_n^{(s)}(\vec{x}) \approx u_n^{(c)}(\vec{x}) := \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{|\vec{l}|_1 = n+d-1-q} u_{\vec{l}}(\vec{x}). \tag{2.21}$$

The underlying function space of the *combined solution* $u_n^{(c)}$ is the sparse grid space

$$\sum_{|\vec{l}|_1 \leq n+d-1} V_{\vec{l}} = \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}} = V_n^{(s)}. \tag{2.22}$$

Thus, for interpolation (with the standard linear basis, as used in this work) it holds $u_n^{(s)} = u_n^{(c)}$. However, for other problems, as the solution of a PDE, this equivalency usually does not hold [39]. Figure 2.6 shows an example for the classical combination technique. Here, it can easily be seen that the union of the component grids results in the sparse grid on the right.

Provided that f fulfills the smoothness conditions presented in Section 2.4, the asymptotic decrease of the approximation error $f - u_n^{(c)}$ is the same as for sparse grids. Since there are grid points which occur multiple times in different component grids, the total number of grid points being computed is higher for the combination technique than on the corresponding sparse grid. There are $\mathcal{O}(d \log(h^{-1})^{d-1})$ component grids with $\mathcal{O}(2^n)$ grid points each. However, asymptotically the number of grid points still is significantly lower than on the corresponding full grid.

The classical combination technique is only formulated for the case $\vec{n} = n \cdot \vec{1}$, where $\vec{1} = (1, \ldots, 1)^T$. A general *combination scheme* can be given as

$$u_{\mathcal{I}}^{(c)}(\vec{x}) = \sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} u_{\vec{l}}(\vec{x}) \,, \tag{2.23}$$

where the index set $\mathcal{I}$ specifies the set of level vectors corresponding to the component grids used in the scheme. The corresponding sparse grid approximation space is defined as

$$V_{\mathcal{I}}^{(s)} = \sum_{\vec{l} \in \mathcal{I}} V_{\vec{l}} = \bigoplus_{\vec{l} \in \mathcal{I}} W_{\vec{l}}. \tag{2.24}$$

The set $\mathcal{I}$ can be arbitrarily chosen under the condition that an admissibility criterion is fulfilled [31]: for each $\vec{l} \in \mathcal{I}$ it must hold

$$\vec{l} - \vec{e}_k \in \mathcal{I} \quad \text{for } 1 \leq k \leq d,\ l_k > 1, \tag{2.25}$$

where $\vec{e}_k$ is the unit vector in direction k. Then, the *combination coefficients* $c_{\vec{l}} \in \mathbb{R}$ can be determined [76] by

$$c_{\vec{l}} = \sum_{\vec{z} = \vec{0}}^{(1,\ldots,1)} (-1)^{|\vec{z}|_1} \cdot \chi^{\mathcal{I}}(\vec{l} + \vec{z}), \tag{2.26}$$

where

$$\chi^{\mathcal{I}}(\vec{l} + \vec{z}) = \begin{cases} 1 & \text{if } \vec{l} + \vec{z} \in \mathcal{I} \\ 0 & \text{else.} \end{cases} \tag{2.27}$$

In this setting there exist various approaches to choose the combination scheme for the problem at hand. A common variant of the classical combination technique is to exclude component grids that do not fulfill a desired minimum level $\vec{l}_{\min}$ from the combination scheme, i.e.,

$$\mathcal{I}_{\vec{l}_{\min}} = \{l \in \mathcal{I} : \vec{l} \geq \vec{l}_{\min}\}. \tag{2.28}$$

This might be necessary, e.g., for the solution of PDEs in order to exclude discretizations that are not numerically stable or even produce physically meaningless results. This

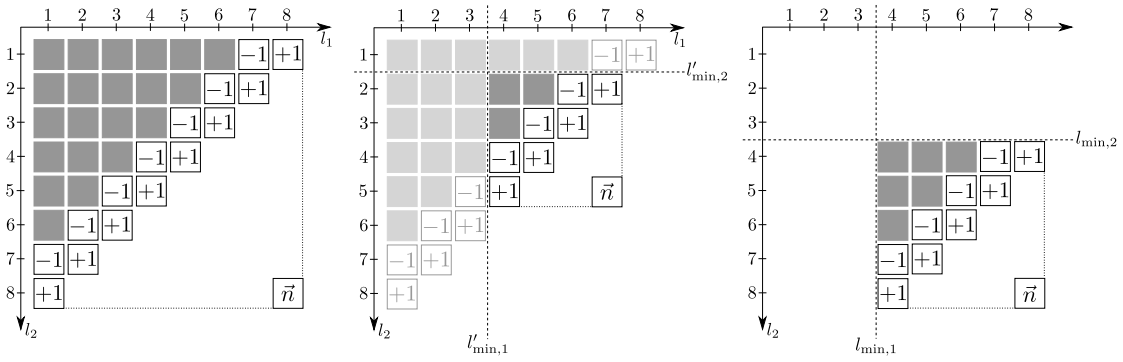

Figure 2.7.: Comparison of different variants of the classical combination technique. The numbers indicate the combination coefficients. The dark grey boxes indicate that the corresponding component grid belongs to the index set $\mathcal{I}$, but has coefficient zero. The box labeled with $\vec{n}$ indicates the function space of the underlying sparse grid $V_{\vec{n}}^{(s)}$. **Left:** The classical combination scheme for $n = 8$. **Middle:** The classical combination scheme for $n = 8$ was truncated, i.e., all component grids below the minimum level $\vec{l}'_{\text{min}}$ were removed from the index set. **Right:** A truncated combination scheme with target level $(8, 8)$. Here, the smallest possible $\vec{l}_{\text{min}}$ was chosen which fulfills $\vec{l}_{\text{min}} \geq \vec{l}'_{\text{min}}$.

approach is called *truncated* combination technique [12]. Note that removing component grids from the combination will change the underlying sparse grid approximation space. However, often it is desirable to obtain a sparse grid approximation of a certain *target level* $\vec{n}$. This is achieved with the set

$$\mathcal{I}_{\vec{n},\vec{l}_{\text{min}}} = \{\vec{l} \in \mathbb{N}^d : |\vec{l}|_1 \leq |\vec{l}_{\text{min}}|_1 + c,\ \vec{l}_{\text{min}} \leq \vec{l} \leq \vec{n}\}, \tag{2.29}$$

where $\vec{l}_{\text{min}} = \vec{n} - c \cdot \vec{1}$ and $c \in \mathbb{N}_0$. For an arbitrary choice of the minimum level $\vec{l}'_{\text{min}}$, c and $\vec{l}_{\text{min}} \geq \vec{l}'_{\text{min}}$ are determined by

$$c = \min_k (n_k - l'_{\text{min,k}}). \tag{2.30}$$

The classical combination scheme and the two variants are visualized in Figure 2.7. In the middle, one can see how the truncation results in a smaller approximation space of the underlying sparse grid. On the right, the desired approximation space was restored with the combination scheme formulated in Equation (2.29). Obtaining the same sparse grid space with a truncated combination technique results in fewer, but larger component grids. Note that $\mathcal{I}$ usually contains a number of component grids where the combination coefficient is zero. Of course, these grids do not actually have to be computed and could be removed from the combination scheme after the combination coefficients have been determined.

There also exist adaptive versions of the combination technique [52, 32, 29]. Here, the idea is to refine the index set $\mathcal{I}$ in multiple steps until the combined solution fulfills

a desired error criterion. In each step the component grid is added to $\mathcal{I}$ which, from a set of possible candidates, is estimated to be most beneficial for the approximation quality of the combined solution. Another variant of the combination technique is the *optimized combination technique* (opticom) [53]. Here, the combination coefficients for a given index set $\mathcal{I}$ are not determined by (2.26). Instead the optimal coefficients that minimize the error of the combined solution with respect to a problem specific norm are determined by solving an additional optimization problem. It has been applied, e.g., to regularized regression problems [28] and to eigenvalue problems [30].

Although the combination technique is more costly with respect to the number of grid points that have to be computed, it has certain advantages over the direct sparse grid approach. In the context of this work, the most outstanding advantage of the combination technique is that it is inherently parallel: all the component grids can be computed in parallel, independently of each other, and the computation of each component grid can be parallelized as well. Thus it is a promising approach to tackle the scalability challenges addressed in this work. Parallelization of the direct sparse grid approach is far more difficult due to the non-local support of the basis functions. Although efficient implementations exist that use shared-memory parallelization, for the problems addressed in this work, where the data is distributed over many compute nodes, the direct sparse grid approach is not well-suited. Another important advantage of the combination technique is that in many cases, existing application codes can be reused. In the best case the application code can be treated as a black box which is used to solve the problem on a given component grid without any substantial modifications to the code itself. This black-box approach is pursued for the development of the combination technique framework presented in this work in order to provide an implementation that can be used for a wide range of application codes. With the direct sparse grid approach, reuse of existing application codes is hardly possible. As most codes that solve PDEs do not use a hierarchical basis, this would require elementary modifications. For a legacy code like GENE, which has grown over more than a decade, these modifications would require enormous personal efforts. The direct sparse grid approach is advantageous especially for problems where spatial adaptivity is beneficial and where distributed memory parallelization is not necessary. Although there exist approaches [40] to introduce some degree of spatial adaptivity for the combination technique via block-structured grids, with the direct sparse grid approach spatial adaptivity is enabled in a more flexible and natural fashion [73, 74].

2.6. Time integration

So far, the spatial discretization of the simulation domain with sparse grids has been discussed, either directly or with the combination technique. For time-dependent problems the question rises how the temporal discretization is done with these methods.

For the direct sparse grid approach, one applies a problem-specific operator $\mathcal{F}$ which evolves the sparse grid solution by a certain time step Δt

$$u^{(s)}_{\vec{n},t+\Delta t} = \mathcal{F}\left\{u^{(s)}_{\vec{n},t}\right\}. \tag{2.31}$$

For the combination technique, a time step of the combined solution is performed by applying an individual operator $\mathcal{F}_{\vec{l}}$ on each component grid in order to evolve all the component solutions by Δt

$$u^{(c)}_{\vec{n},t+\Delta t} := \sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}}\, \mathcal{F}_{\vec{l}}\, \left\{ \mathcal{P}_{\vec{l}}\, (u^{(c)}_{\vec{n},t}) \right\}. \tag{2.32}$$

The operator $\mathcal{P}_{\vec{l}}$ is a projection of the combined solution at the time t into the approximation space $V_{\vec{l}}$ of the corresponding component grid. Note that in order to just introduce this concept, here it is not necessary to further specify the abstract operators $\mathcal{F}$ and $\mathcal{P}$. A more detailed description is provided in later parts of this work.

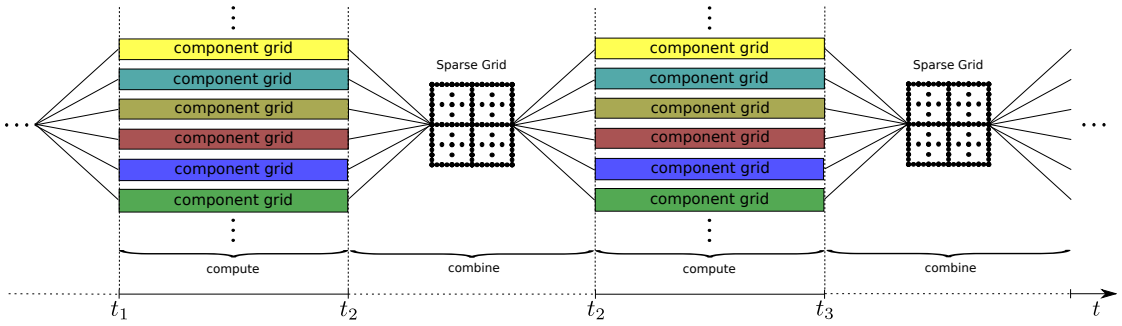

Figure 2.8.: Concept of time stepping with the combination technique. On each component grid, the PDE is solved for a certain time interval Δt. Then the component grids are combined in the sparse grid space, and the combined solution is set as the new initial value for each component grid for the computation of the next time interval.

The concept of time stepping with the combination technique is visualized in Figure 2.8. An initial value problem, e.g., a PDE, is solved for a certain time interval on the component grids. Then the combined solution is evaluated in the sparse grid approximation space. This combined solution is projected back onto the individual component grids and set as the initial values for the computation of the next time interval. This procedure is repeated until a desired stopping criterion is fulfilled, for example, until a predefined point in time is reached. Throughout this work the step in-between two subsequent computation steps is called *combination step*. This approach has been used in [38, 40, 66] for the simulation of fluid flow. An numerical analysis of the combination technique applied to a time-dependent advection equation is presented in [68].

This concept allows to use different time step sizes: an inner time step size Δt for the computation of the component grids and an outer time step size $\Delta T = \#\text{steps} \cdot \Delta t$ for the combined solution, where #steps denotes the number of inner time steps computed on the component grids in-between two combination steps. In fact, the inner time step size and the number of inner time steps can even be chosen individually for each component grid. Throughout this work the outer time step size is usually referred to as *combination interval*. The actual choice of inner and outer time step strongly depends on the specific

application at hand. For some applications it might be enough to combine the component grids only once in the end. For other applications small outer time step sizes are necessary to guarantee convergence and stability of the combined solution. However, even in cases where this is not an issue, it might be desirable to compute the combined solution every few time steps in order to track the development of the solution field over time.

Unfortunately, this approach limits the parallelism of the combination technique, because the component grids cannot be computed independently and asynchronously of each other any more. Now there are points of global synchronization (all processes must wait until all other processes are finished with the current computation step) and communication (data must be exchanged between processes to evaluate the combined solution). Thus, for the large-scale simulations addressed in this work, where the simulation data of each component grid is distributed over thousands of processes of an HPC system, an efficient and scalable implementation of the combination step is most crucial for the overall performance of computations with the combination technique.

2.7. Investigation of different combination strategies

In this section different strategies for the combination step are introduced and analyzed with respect to their theoretical costs. The goal is to identify the strategy which is most promising to scale to very large problem sizes. This means that with increasing problem size the computational effort for the combination step must not grow faster than the overall effort for the computation of the component grids.

This analysis shows that only the combination in the hierarchical basis is suitable for large-scale computations. The combination in the hierarchical basis has been further analyzed and improved in close collaboration with Philipp Hupp, Riko Jacob and Dirk Pflüger [59, 58]. The most relevant findings of these works are summarized in Section 2.7.2.

The strategies presented in the following are formulated for the case of distributed memory systems, as only these kinds of systems are applicable for the large-scale simulations targeted in this work. Furthermore, it is assumed that each component grid is computed by only one compute node of the distributed memory system. This assumption simplifies the description and analysis of the presented strategies. However, all the observations presented here are equally true for the general case where each component grid is computed by multiple nodes.

The presented strategies address the combination step as presented in Section 2.6. The goal is that at the end of this step a projection of the combined solution is available on each component grid. Conceptually the combination step can be expressed in two substeps: a *reduction step*, where the component solutions are reduced to the combined solution; and a *scatter step*, where the combined solution is scattered back to the component grids (compare Figure 2.8).

Although the reduction step is closely related to the evaluation of the combined solution, plain evaluation is not the focus here and thus some of the presented strategies are not directly applicable for this case. One of the reasons is that these strategies are not necessarily formulated as a reduction and a scatter step. Furthermore, sometimes

optimizations are used that cannot be employed when evaluation is the goal. Nevertheless, many of the concepts and observations presented here help to formulate efficient algorithms for the evaluation of the combined solution as well. One such algorithm is presented in Section 3.4.

2.7.1. Costs of different combination strategies

The following analysis considers two types of costs separately. The computational costs that are related to arithmetic operations and main memory transfers within a node and communication costs that are caused when data has to be transferred between compute nodes. Furthermore, these costs are presented from the view of one node. Although the nodes work in parallel, what eventually defines the runtime of the combination step is the runtime of the slowest node. Thus the complexity of the operations of any node is representative for the complexity of the combination step.

Interpolation onto the full grid. A straightforward way to obtain the combined solution is to interpolate the component solutions to the approximation space of the full grid V_n and add them up there. This can be achieved by creating a temporary full grid Ω_n on each node and by interpolating the component solution onto this grid, e.g., by linear interpolation. These temporary grids are then reduced to the combined solution, either by sending all the temporary grids to one node, which then adds them up locally, or by some more sophisticated reduction schemes. Note that for this analysis the details of these operations are not of interest. The costs (arithmetic operations and memory transfers) on each compute node to interpolate the component grid into V_n are proportional to the size of the full grid, i.e., $\mathcal{O}(2^{nd})$. Likewise, and independent of the actual implementation of the reduction, the amount of data that is sent or received by each process is of $\mathcal{O}(2^{nd})$, as well. The obvious drawback of this method is that it reintroduces the complexity of the full grid (the explained goal of sparse grids and the combination technique to avoid the full grid complexity). Thus, for the large-scale simulations addressed in this work such a strategy is not feasible.

Pair-wise interpolation of the component grids. The complexity of the full grid can be avoided by formulating the combination as

$$\mathcal{P}_{\vec{l}'}\{u_n^{(c)}(\vec{x})\} = \sum_{\vec{l}\in\mathcal{I}} c_{\vec{l}}\mathcal{P}_{\vec{l}'}\{u_{\vec{l}}(\vec{x})\}, \tag{2.33}$$

where the abstract operator $\mathcal{P}_{\vec{l}'}$ indicates a suitable projection of the solution from the function space $V_{\vec{l}}$ into the space $V_{\vec{l}'}$. This means that the combined solution projected into the space $V_{\vec{l}'}$ of the component grid $\Omega_{\vec{l}'}$, e.g., by linear interpolation, can be formulated as a sum of projections of the other component grids into $V_{\vec{l}'}$. Of course, this has to be done with each component grid and so one can say each pair of component grids has to be projected into the space of the respective other component grid. For each component grid this results in $|\mathcal{I}| \in \mathcal{O}(d\log(h^{-1})^{d-1})$ projections that have computational costs in $\mathcal{O}(2^n)$

(size of the component grids). Independent of whether the projections are computed first and then sent to the respective node or the other way around, the amount of data received by each node is in $\mathcal{O}(d \log(h^{-1})^{d-1}) \times \mathcal{O}(2^n)$. However, with this strategy the complexity of the combination step is still higher than the computation of the component grid, which is in $\mathcal{O}(2^n)$.

Interpolation onto the sparse grid. A straightforward improvement of the interpolation onto the full grid is to interpolate each component grid into the underlying sparse grid space $V_n^{(s)}$ and to add up the component solutions there. Again, this can be achieved by a simple linear interpolation operator. The cost for the interpolation and the amount of data to be sent and received by each node has the complexity of the size of the sparse grid $\mathcal{O}(2^n \log(2^n)^{d-1})$. Thus, although significantly lower than interpolation to the full grid, with this approach the complexity is still higher than for the computation of the component grid.

Combination in the hierarchical basis. In this variant, the combined solution is not obtained by interpolation, but instead the component grids are combined in the hierarchical basis of the sparse grid $V_n^{(s)}$ by adding up the hierarchical coefficients. This variant is motivated by the observation that for each component grid $\Omega_{\vec{l}'}$ in the hierarchical basis representation only the hierarchical subspaces $W_{\vec{l}}$ with $\vec{l} \leq \vec{l}'$ have non-zero coefficients. The advantage of this variant is that the other subspaces in $V_n^{(s)}$ with zero coefficients do not need to be sent to other nodes, as they do not contribute to the combined solution. Thus the amount of data to be sent or received by each node reduces to $\mathcal{O}(2^n)$ (size of the component grid). The computational costs for the hierarchization and dehierarchization of a component grid are in $\mathcal{O}(2^n)$, as well. This strategy for the combination step thus is the most promising to scale to large-scale problem sizes, as it is the only one which has the same complexity as the computation of the component grids. It will be further analyzed in the following section.

2.7.2. Optimal communication schemes for the combination in the hierarchical basis

Having identified the combination in the hierarchical basis to be the most promising approach for an efficient implementation of the combination step, this strategy will be further analyzed in the following. The analysis focuses on the communication part, after the component grids have been hierarchized. As the communication part of the combination step introduces a global communication and synchronization bottleneck, efficient and scalable solutions are crucial to enable scalability to large problem sizes.

An important observation that can be made for the combination in the hierarchical basis is that for each hierarchical subspace $W_{\vec{l}} \in V_n^{(s)}$ there is a different subset $\mathcal{C}(W_{\vec{l}})$ of the index set $\mathcal{I}$ that corresponds to the component grids which share this subspace

$$\mathcal{C}(W_{\vec{l}}) = \left\{ \vec{l}' \in \mathcal{I} : \vec{l}' \geq \vec{l} \right\}. \tag{2.34}$$

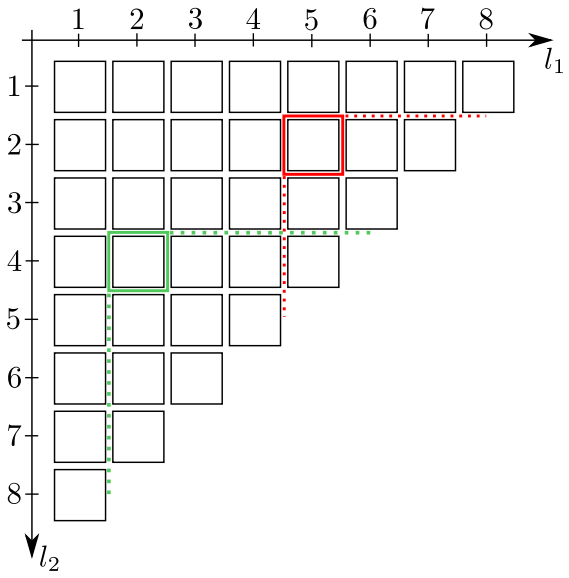

Figure 2.9.: Two subsets, $\mathcal{C}(W_{(5,2)})$ (red) and $\mathcal{C}(W_{(2,4)})$ (green), of the index set $\mathcal{I}$ are marked. The component grid corresponding to $\vec{l} = (5, 4)$ is contained in both sets.

This is visualized in Figure 2.9.

This observation can be exploited for the design of optimal communication schemes:

- Each hierarchical subspace $W_{\vec{l}}$ has to be exchanged only between the compute nodes responsible for the component grids which share this subspace, i.e., the component grids corresponding to $\mathcal{C}(W_{\vec{l}})$.
- There are hierarchical subspaces $W_{\vec{l}}$ which only exist in one component grid, i.e., $|\mathcal{C}(W_{\vec{l}})| = 1$. These subspaces do not have to be exchanged at all.

On this basis, different communication schemes were proposed.

Algorithm 1: Sparse Grid Reduce

```
1 create buffer B
2 for each W_l ∈ V_{n-1}^{(s)} do
3     if W_l ∈ V_l then
4         append W_l to B
5     else
6         append zeros to B
7 AllReduce(B,I)
8 for each W_l ∈ V_l do
9     extract W_l from B
```

Sparse Grid Reduce. *Sparse Grid Reduce* is the straightforward communication scheme for the combination in the hierarchical basis. The algorithm is described in Algorithm 1. First each component grid is hierarchized. Then a temporary sparse grid data structure is created on each node and the hierarchical coefficients of the component grid are copied to their respective place in this data structure. Hierarchical subspaces of the sparse grid that are not contained in the component grid are simply set to zero. Then this sparse grid is reduced and the result is scattered back to the individual nodes by the so-called *AllReduce* routine (an explanation of this routine follows later). It is assumed that an efficient implementation of *AllReduce* is available on the systems for which these communication schemes are designed. Afterwards, for each component grid the hierarchical coefficients are extracted from the sparse grid data structure and copied back to the right position in the component grid. Then the component grids are dehierarchized in order to conclude the projection of the combined solution into the function space of the component grid. The function space of the sparse grid used for the reduction is $V_{n-1}^{(s)}$. This way, it can be avoided to transfer the largest hierarchical subspaces in $V_n^{(s)}$ (the sparse grid corresponding to the combination scheme) which always exist only in one component grid. Compared to a strategy which is based on interpolation to the (full) sparse grid, this reduces the amount of data to be transferred already by almost a factor of two. Although this communication scheme is not optimal with respect to communicated data volume, for real implementations this can be the best choice in some situations, because it requires only a single *AllReduce* operation. This will be further discussed in the following.

Algorithm 2: Subspace Reduce

```
1 for each W_l⃗ ∈ V_{n-1}^{(s)} do
2     if W_l⃗ ∈ V_l⃗ then
3         copy W_l⃗ into buffer B
4         AllReduce (B, C(W_l⃗))
5         extract W_l⃗ from B
```

Subspace Reduce. The idea of *Subspace Reduce* is to reduce each hierarchical subspace of the sparse grid individually using the *AllReduce* Operation. For each subspace $W_{\vec{l}}$ only the nodes which store a component grid containing this subspace (i.e., a component grid in $\mathcal{C}(W_{\vec{l}})$) contribute to *AllReduce*. The algorithm is described in Algorithm 2. With this strategy each node sends and receives only the absolutely necessary amount of data. Compared to *Sparse Grid Reduce* no subspaces are sent that contain only zeros. Also, no subspaces have to be received that are not contained in the component grid of the node.

Parallel Subspace Reduce. The third communication scheme is rather a variant of *Subspace Reduce* than a strategy on its own. It does not try to achieve a lower amount of communicated data than *Subspace Reduce* (as this is already optimal), but instead it tries to reduce the runtime of the combination step by enabling more parallelism for

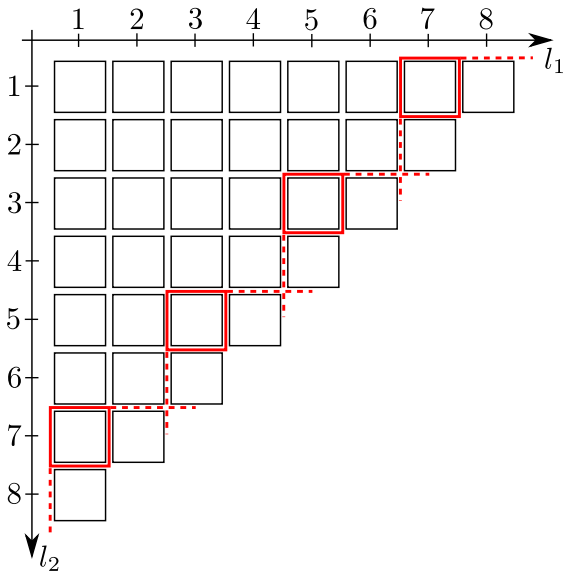

Figure 2.10.: Concept of *Parallel Subspace Reduce*. The component grids marked with solid red lines can be reduced in parallel.

the execution of Algorithm 2. This algorithm can be implemented using MPI (*Message Passing Interface*), the most widely-used programming model for distributed memory systems, exactly as presented here. Each process iterates over the subspaces $W_{\vec{l}}$ in the same prescribed order. If a process reaches the *AllReduce* function for a specific subspace it must wait until all processes that contain this subspace (that store the component grid corresponding to $\mathcal{C}(W_{\vec{l}})$) reach this function as well. This means that the subspaces are processed in a sequential fashion, one after the other, and at each point in time only a subset of the processes do some actual work, those who contribute to the *AllReduce* operation. As visualized in Figure 2.10, there are, however, subspaces $W_{\vec{l}}$ for which the corresponding sets $\mathcal{C}(W_{\vec{l}})$ are disjoint. For these subspaces the *AllReduce* can be executed by different sets of processes and thus more than one subspace can be reduced at the same time. The idea of *Parallel Subspace Reduce* is to specify an order in which the subspaces are reduced such that as many subspaces as possible can be reduced at the same time. The algorithm presented in [58] iterates over the diagonals of the index set $\mathcal{I}$ and identifies the subspaces with disjoint sets for each diagonal. The a-th diagonal in the combination scheme is defined by the index set

$$\mathcal{D}_{\mathcal{I},a} := \left\{ \vec{l} \in \mathcal{I} : |\vec{l}|_1 = a + d - 1 \right\}, \tag{2.35}$$

where $1 \leq a \leq n$. Although, this does not achieve the optimal degree of parallelism, it is relatively simple to analyze and implement this strategy. One can observe that the higher the discretization level of a hierarchical subspace, the lesser the number of component grids containing this subspace. Furthermore, as the size of the subspaces increases exponentially with the discretization level, not only does the parallelism increase

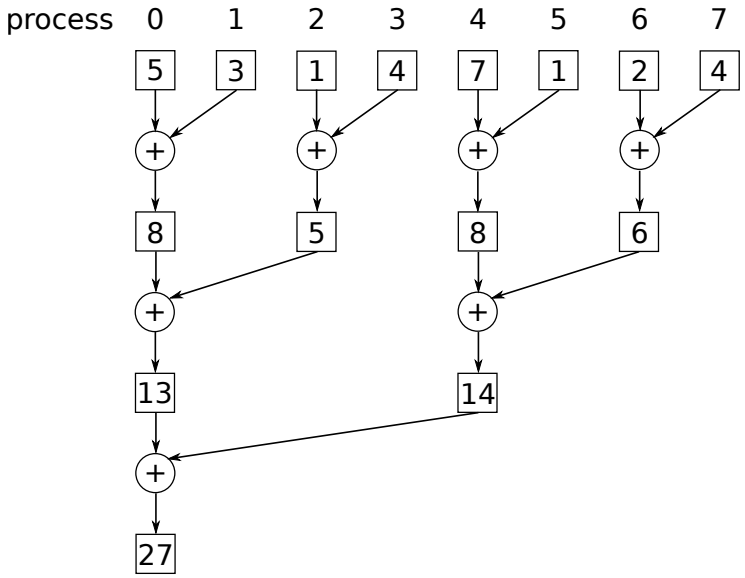

Figure 2.11.: Concept of parallel reduction.

for larger diagonals (higher parameter a), but also the benefit in runtime compared to the serial *Subspace Reduce*.

All of the above presented communication schemes base on the operation *AllReduce*. *AllReduce* solves the following task: Initially all compute nodes contributing to this operation hold their own version version $\vec{v}_i$ of a vector $\vec{v}$. After the execution of *AllReduce* each compute node holds the result

$$\vec{v}_{\text{res}} := \sum_{i=1}^{m} \vec{v}_i, \tag{2.36}$$

where m is the number of compute nodes. This can be solved in two phases, a *reduction* operation followed by a *scatter* operation. The communication structure of the reduction is motivated by binomial trees: each node sends its version of the vector to its parent node in a underlying binomial tree, where it is then added up. This is visualized in Figure 2.11 at the example of scalar values. The communication structure of the scatter operation is just inverse to the reduction.

A simple standard communication model is used to analyze these operations, similar to the models typically used to analyze the performance of collective operations in MPI [82]. In this model the communication is synchronous and round-based. In each round, every node can either send a message to exactly one other node or receive a message. The duration of a round is determined by the time it takes to send the largest message. All nodes are uniformly connected, i.e., each node can send a message to each other node

with the same bandwidth (transfer rate) B. The time to send a message of size N is then

$$L + \frac{N}{B}, \tag{2.37}$$

where the latency L is a time which is independent of the message size, e.g., the time to establish the connection.

In this model using a reduction step and a gather step based on binomial trees for *AllReduce* is optimal with respect to the number of rounds, the number of messages and the total communicated volume. The reduction and the scatter operation each require $\lceil \log_2(m) \rceil$ rounds. Thus, the total communication time for *AllReduce* is

$$2\lceil \log_2(m) \rceil \left(L + \frac{N}{B} \right). \tag{2.38}$$

Note that actual implementations of *AllReduce* do not necessarily employ two distinct phases and binomial trees. Assuming bidirectional communication networks (a node can send and receive a message at the same time), the number of rounds can be reduced from $2\lceil \log_2(m) \rceil$ to $\lceil \log_2(m) \rceil$ by an approach called *recursive-doubling* [82]. In fact, actual implementations might also switch between different algorithms depending on the size of $\vec{v}$ and the number of communication partners. However, taking all of this into account would make the analysis of the communication schemes prohibitively difficult. Nevertheless, this does not make a difference for the design of the algorithms. As the proposed algorithms base on *AllReduce*, a better implementation of this operation would also be beneficial for the performance of the algorithms.

Using this model to analyze the combination schemes, it was found that *Sparse Grid Reduce* is optimal with respect to the number of messages and the number of rounds, whereas *Subspace Reduce* is optimal with respect to the total communicated volume. The number of messages and the total volume of *Parallel Subspace Reduce* is identical to *Subspace Reduce*, but due to the parallelism *Parallel Subspace Reduce* can be executed in a lower number of rounds. Another important characteristic is the so-called *makespan volume*. This is the maximal communicated volume within each round (the time it takes to transfer the largest message within a round defines the duration of the round) summed over all rounds. Thus, the makespan volume is closely related to the total execution time of the different combination schemes and therefore it is the most meaningful characteristic to compare their performance. Here, it shows that there is no communication scheme which has the lowest makespan volume for all problem configurations, but rather that there is a trade off between sending few large messages (*Sparse Grid Reduce*) and sending many small messages (*Subspace Reduce*). However, the makespan volume of *Parallel Subspace Reduce* is always lower than that of *Subspace Reduce*. The detailed analysis is given in [58].

The performance of the different communication schemes was investigated by experiments on the supercomputers *Hermit* and *SuperMUC Phase 1*. See Appendix A for a short description of the systems. The investigations comprised two components. A runtime model based on the models presented above was used to predict the performance

of the different communication schemes. The model was then verified by measurements. In order to determine the model parameters latency L and bandwidth B, a so-called ping-pong test was performed. Sending an MPI message back and forth between two nodes and measuring the duration allows to estimate the latency and the bandwidth of the network connection between nodes according to Equation (2.37). However, the estimates obtained this way strongly depend on the actual size of the message and the position of the two nodes in the network topology. Thus, with this procedure one obtains upper and lower bounds for latency and bandwidth, rather than unique values. Using these upper and lower bounds for latency and bandwidth resulted in an upper and lower bound for the runtime model.

The actual implementation of the communication schemes is very close to the algorithms presented above. For the *AllReduce* operation the function `MPI_Allreduce` was used. The different sets $\mathcal{C}(W_{\vec{l}})$ were represented by different MPI communicators. Additionally, variants of *Subspace Reduce* and *Parallel Subspace Reduce* were used for the experiments that employ the non-blocking `MPI_Iallreduce`. Here, the algorithm does not wait until the current subspace $W_{\vec{l}}$ is processed, but directly proceeds with the next subspace. The actual communication is handled in the background by the MPI runtime system and the network controllers. This allows the system to rearrange the communication structure on a lower system level with a fine granularity in order to optimize the utilization of the communication channels. Of course, this violates the assumptions of the round-based model. However, these variants potentially show a better performance than their blocking counterparts. Unfortunately, at the time when the experiments were performed, the MPI implementation of *SuperMUC* did not yet support `MPI_Iallreduce`. Thus, experimental results with these variants could only be obtained on *Hermit*. In accordance with the assumptions for the model, for the experiments each component grid was stored on a single node and a single MPI process was used for the communication.

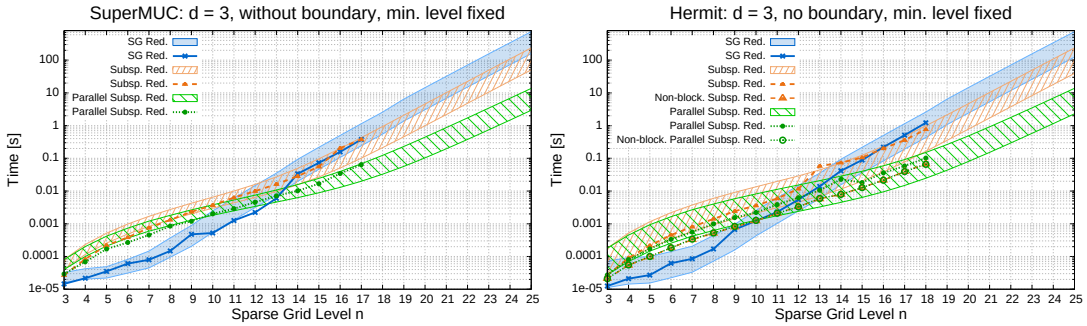

Figure 2.12.: Results on SuperMUC **(left)** and Hermit **(right)** for $d = 3$ without boundary. $\vec{l}_{\min}$ was fixed to (1,1,1). Thus, when n was increased the number of component grids (and compute nodes) increased.

Figure 2.12 shows results in three dimensions on the two systems. The number of nodes used for this experiment ranged from 10 to 460. The results are similar on both systems. For small problem sizes *Sparse Grid Reduce* was fastest. Here, the hierarchical

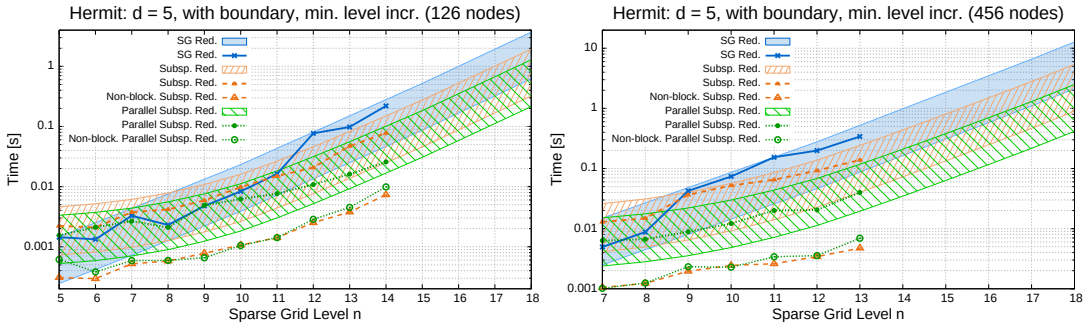

Figure 2.13.: Results on Hermit for $d = 5$ with boundary. **Left:** When n was increased, $\vec{l}_{\min}$ was increased accordingly from $(1, 1, 1, 1, 1)$ for $n = 5$ to $(3, 3, 3, 3, 2)$ for $n = 14$ in order to obtain a constant number of 126 component grids. **Right:** When n was increased, $\vec{l}_{\min}$ was increased accordingly from $(1, 1, 1, 1, 1)$ for $n = 7$ to $(3, 2, 2, 2, 2)$ for $n = 13$ in order to obtain a constant number of 456 component grids.

subspaces are very small (without boundary they are especially small) and so there is not much overhead for *Sparse Grid Reduce*. However, the time of *Sparse Grid Reduce* increased faster with increasing n and so at some point this method performed worse than the others. *Parallel Subspace Reduce* was always faster than the simple *Subspace Reduce*. Furthermore, the difference between the two methods increases with n, because with a larger number of increment spaces also more parallelism is possible. The colored areas in the plots show the predictions of the runtime model (for each method an upper bound and lower bound is given). One can observe that the model captures the trends of the experimental results quite well. On *Hermit* also the results of the non-blocking variants of *Subspace Reduce* are presented. They were fastest over the whole range of n. Hence, the non-blocking variants of `MPI_Allreduce` were beneficial for the performance of the algorithms. As there was no visible difference between the non-blocking versions of *Subspace Reduce* and *Parallel Subspace Reduce*, one can conclude that rearranging the order of the subspaces, as it is done in *Parallel Subspace Reduce*, does not have a performance benefit here.

Figure 2.13 shows results in five dimensions on Hermit. Here, for each iteration of n the minimum level was adapted so that a constant number of 126 nodes (left) or 456 nodes (right) was used. The behavior is similar as observed for three dimensions. For small problem sizes *Sparse Grid Reduce* performs well, but with increasing n the runtime grows significantly faster than for the other methods. This happens earlier with 456 nodes, because here for the same n the size of the underlying sparse grid is larger than with 126 nodes (due to the larger $\vec{l}_{\min}$) and thus the overhead for *Sparse Grid Reduce*. Here, the non-blocking variants clearly outperform all other methods, even for small problem sizes.

A detailed description of the experiments, more experimental results and more details on the runtime model and the implementation of the communication schemes can be

found in [58]. One advantage of the runtime model is that it can also be used to predict the duration of the communication step for problem sizes where it is out of scope to do experiments. Such predictions are also included in [58].

The above investigations demonstrate that the combination in the hierarchical basis allows to optimize the communication structure of the combination step. With *Sparse Grid Reduce*, already a significant reduction in communicated volume (almost a factor of two) can be achieved compared to the case where the whole sparse grid is exchanged (including the largest hierarchical subspaces) as it would be required when interpolation onto the sparse grid is used for the combination. With the more sophisticated approaches *Subspace Reduce* and *Parallel Subspace Reduce*, the time for the communication can be even further reduced, especially for large problem sizes and when non-blocking MPI calls are used.

3. Development of a massively parallel combination technique

The main contribution of this work was the development of new algorithms that enable the massively parallel computation of high-dimensional PDEs with the combination technique. It is the first implementation of a parallel combination technique that was demonstrated to enable scalability for time-dependent and high-dimensional PDEs on a full supercomputer. It enables the computation of extremely high spatial resolutions that would be out of reach with the direct full grid approach.

The goal was to create a general software framework for the parallel combination technique which is independent of a specific application and thus can be reused for future experiments with other application codes than GENE. In order to achieve this, the design follows a black-box approach where the functionality of the actual application code is abstracted by a high-level interface. This interface has to be adapted by the user to the specific application in order to provide an efficient exchange of data between the framework and the application. The framework forms a part of the sparse grid software library SG++ [4].

In this chapter the basic components of this framework are introduced and analyzed:

- a scalable parallelization concept which exploits the two levels of parallelism of the combination technique,
- effective load balancing schemes, and
- efficient and scalable algorithms for the combination step that enable the computation of time-dependent problems.

Furthermore, the software framework is also capable of algorithm-based fault-tolerance (see Chapter 4).

3.1. Parallelization concept

The combination technique offers two levels of parallelization:

- *A fine parallelization level* – each component grid can be computed in parallel.
- *A coarse parallelization level* – all the component grids can be computed in parallel and individually of each other.

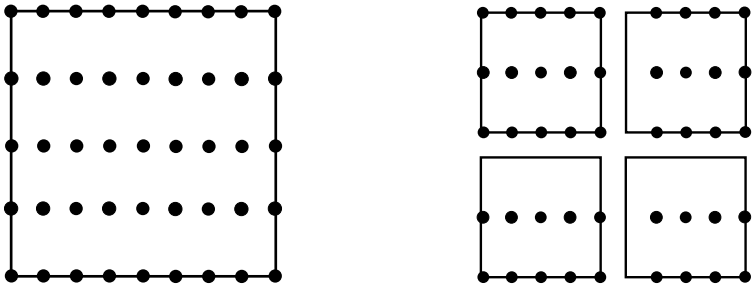

Figure 3.1.: Left: Component grid with $\vec{l} = (3, 2)$ and boundary points. Right: The same component grid distributed onto four processes. The boxes indicate the local subdomains for each process.

The fine parallelization level relates to the parallelization scheme of the application code being used to compute the component grids. For grid-based discretizations the most natural distributed-memory parallelization scheme is to geometrically decompose the problem domain into equally sized subdomains. Then, each process of the parallel application is responsible to compute and store only the grid points in its own local subdomain. For the regular grids used in this work this results in about the same number of grid points on each process. The data structure that represents this domain decomposition in the software framework is the so-called *distributed component grid*. Figure 3.1 shows an example with four processes. This data structure represents a geometric domain decomposition where the processes are arranged in a Cartesian grid as well. For certain domain decompositions (e.g., the number of processes in each dimension are powers of two) there are grid points positioned directly on the subdomain boundaries. For the algorithms used in this work each grid point must be assigned to exactly one process, and so one would assign these grid points to either of the neighboring processes. However, it can happen that the actual domain decomposition and assignment of grid points to the processes in the application code differs from this structure. Therefore, the user needs to provide appropriate methods in the interface class (see Section 3.5) that convert the application data into the format of the distributed component grid before the combination step (and back afterwards).

For the coarse parallelization level the manager-worker pattern is employed. In this concept the available compute resources, in form of MPI processes, are arranged in process groups. A dedicated manager process assigns the component grids to the process groups so that a good load balance is achieved. The manager does not participate in the computations or store any simulation data. This is solely done by the process groups. The role of the manager is to coordinate the individual steps of the parallel combination algorithm. From the point of view of the manager process the component grids are abstracted by so-called tasks. A task does not only consist of the component grid, but rather represents the interface to the application (see Section 3.5). This is illustrated in Figure 3.2. A similar parallelization concept for the combination technique has already been successfully employed in [36].

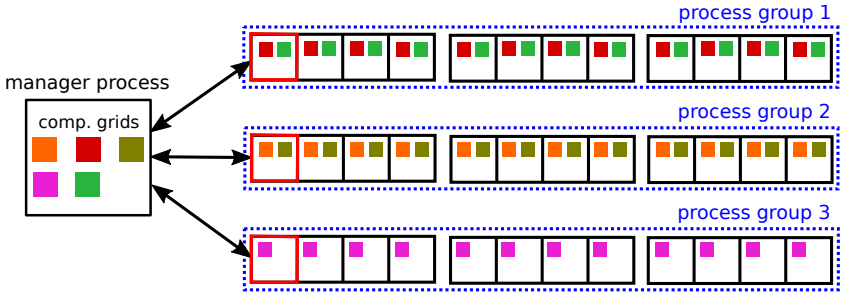

Figure 3.2.: The manager-worker pattern. In this example the parallel resources are arranged in three groups of 12 processes. A manager process assigns the compute tasks to the process groups. In each group only the master process (red) communicates with the manager.

Algorithm 3: Parallel combination technique.

input :
t_0, t_{end}: start and end time
ΔT: outer time step size
$\mathcal{I}$: set of component grids
c_u: combination coefficient corresponding to component grid u
groups: set of process groups
grids(g): set of component grids assigned to group g

```
1  distribute component grids in I to process groups and set initial conditions
2  t_i ← t_0
3  while t_i ≤ t_end do
4      for g ∈ groups do in parallel
5          for u ∈ grids(g) do
6              u ← solve (u, ΔT) ;                          // solve u for next outer time step
7      for g ∈ groups do in parallel
8          dsg_g ← dsg(n⃗) ;                                  // create distributed sparse grid
9          for u ∈ grids(g) do
10             u ← hierarchize (u) ;                        // hierarchize component grid
11             dsg_g ← dsg_g + c_u · u ;                    // add hierarchical coefficients to sparse grid
12     dsg_g ← Σ_{g∈groups} dsg_g ;                         // global reduction of sparse grid
13     for g ∈ groups do in parallel
14         for u ∈ grids(g) do
15             u ← dsg_g ;                                  // extract hierarchical coefficients from sparse grid
16             u ← dehierarchize (u) ;                      // dehierarchize component grid
17     t_i ← t_i + ΔT
```

Algorithm 3 describes how the parallel combination technique is embedded in the manager-worker scheme. For the sake of simplicity, here it is assumed that the component grids were already assigned to the process groups. In this algorithm one can identify the main steps of the combination technique for time-dependent problems. Lines 4 to 6 correspond to the computation of the component grids for one outer time step ΔT. Inside each process group the component grids assigned to the group are processed one

after each other by all processes of the group. The groups, however, work in parallel and independently of each other. Lines 7 to 16 correspond to the combination step. This step is rather complex and described in detail in Section 3.3. However, here one can already identify the main substeps of this step, as introduced in Sections 2.6 and 2.7: *hierarchization*, *reduction to the combined solution* and *dehierarchization*.

The manager-worker pattern is a natural way to represent the two-level parallelism of the combination technique. The component grids are computed completely independent and asynchronously of each other by the process groups they are assigned to. While the component grids are computed, there is no communication between processes of different process groups. Global communication between processes of different groups occurs only during the combination step.

This parallelization concept offers high flexibility. The number of process groups, as well as the number of processes per group can be adjusted to the needs of the application code, the problem size and the underlying compute architecture. The right choice of these parameters lies in-between the two extreme cases:

- using one process group for each component grid, or
- using only one large process group.

Using one process group for each component grid enables the highest degree of parallelism for the computation phase. Unfortunately, this also leads to a massive load imbalance (the runtime of the component grids can be very different) on the coarse parallelization level. Theoretically, load balance can be achieved by choosing exactly the right number of processes for each component grid. However, finding the optimal distribution of processes fails in practice, because usually the models to predict the application's runtime are not exact enough (or do not exist at all) or the application code does not accept all possible numbers of processes for the parallelization (e.g., only powers of two are allowed). Thus, the more robust and promising way to solve the load-balancing problem is to use less process groups than component grids and to employ effective load balancing schemes as described in Section 3.2. Furthermore, the time for the combination step increases with the number of process groups. When the combination step occurs very frequently, this can result in noticeable overhead. See Section 3.3 for further details.

Although using only one process group results in the optimal load balance for the coarse parallelization level, this is not a sensible choice as well, because usually the parallel efficiency for the fine parallelization level decreases with increasing number of processes. Using too less process groups inhibits the potential of the combination technique to increase the overall parallel efficiency by computing multiple moderately sized problems in parallel. Thus, the optimal balance between the number of process groups and the size of the groups strongly depends on the specific application and the compute architecture.

The role of the manager process is to coordinate the different process groups. This is realized by a set of control messages between the process groups and the manager. A process group has different modes of operation that can be modeled by a finite state machine as illustrated in Figure 3.3. The process group changes its state according to the signal it receives from the manager:

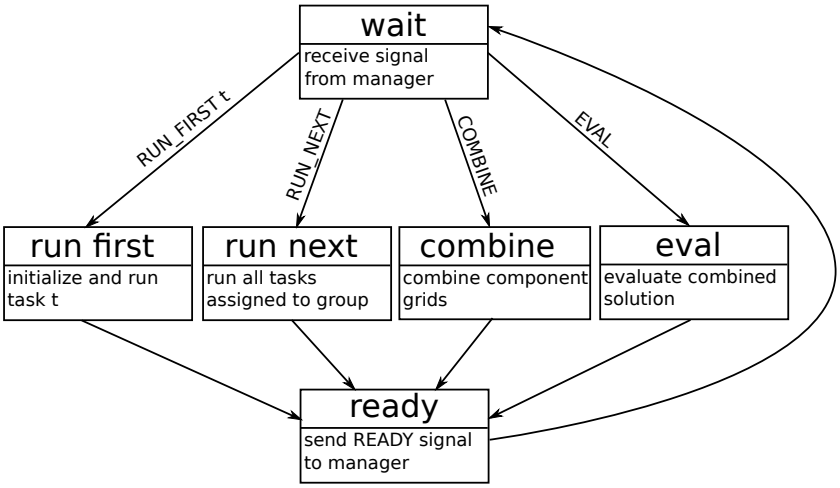

Figure 3.3.: Finite state machine modeling the different modes of operation of a process group.

- *wait*: The process group idles until it receives a signal from the manager.
- *run first*: After receiving the `RUN_FIRST` signal, the process group also receives a task `t`. If the application has a dedicated initialization routine, this is called now. Then, the application code is executed for the given outer time step size. Afterwards, the data structures to store the (distributed) component grid are created and the simulation data is transferred (and maybe converted) from the application to the component grid.
- *run next*: In this state the process group computes all of its task, one after the other. Before the computation of a task is started, the simulation data has to be transferred from the component grid into the application's data structures. After the computation the data is transferred back into the component grid.
- *combine*: In this state the component grids are combined in the sparse grid space. This happens in three stages. First, all the locally available component grids are combined inside the process group. Then, the combined solution is reduced globally across all process groups. In a last step, the combined solution is transferred back onto the component grid of each task. As the combination step involves the global reduction, all process groups must be in this state at the same time.
- *eval*: In this state the process group evaluates the combined solution (which is stored in each process group after it has been in the *combine* state).
- *ready*: After finishing one of the above states the process group changes into the *ready* state. Here, it sends a `READY` signal to notify the manager that it is ready to receive a new signal.

Note that there is an implicit order of theses states. For example, none of the other states is possible if the process group was not in *run first* before and has computed one component grid at least. The manager takes care that the process groups execute these states in a valid order.

The communication between the manager and the process groups is implemented by MPI messages. Each process group has a dedicated master process which receives the message from the manager and forwards it to the other members of the group. Accordingly, only the master process sends the `READY` signal. This communication hierarchy minimizes the number of messages between manager and process groups and avoids a global one-to-all situation, where the manager has to talk to all processes. These control messages do not contain any simulation data and are solely used to coordinate the process groups. Thus, the effort for the communication between manager and process groups is negligible.

3.2. Load balancing

The parallelization concept described above allows for simple but very effective load balancing strategies. Before the actual computations begin, the manager creates a list of tasks according to the chosen combination scheme. The manager sorts this list of tasks in decreasing order of the expected compute times. During the *run first* stage the manager picks the first (the most expensive) tasks from the list and assigns it to a idle process group. Whenever a group receives a task, it directly starts to process it. This is repeated until all process groups are busy. As soon as a process group finishes the computation of its current tasks, it gets assigned with a new task from the list by the manager. This is repeated until all tasks are assigned to the process groups and marks the end of the *run first* stage. A similar approach has been used in [36].

This load balancing strategy was inspired by algorithms for the solution of the bin packing problem, a well-studied problem of scheduling theory [21]. Although this strategy does not guarantee to find the theoretically optimal load-balance (finding the optimal solution is an NP-complete problem), it is a simple and effective practical approach that provides very good load balance on the average.

Sorting the tasks according to their expected compute times, requires a model to estimate these times. The software framework is designed so that an application specific cost model can be provided by the user. A natural cost model for the combination technique is to count the number of grid points of the component grids. This was used in [36]. In [49], we proposed a more accurate cost model and demonstrated that this model improves the load balance over the standard model when a large number of process groups is used.

3.2.1. Cost models

The most natural cost model for the combination technique is to count the number of grid points N^* in a component grid

$$N^*(\Omega_{\vec{l}}) := \prod_{i=1}^{d} (2^{l_i} \pm 1) \approx 2^{|\vec{l}|_1} \tag{3.1}$$

(+1 if the grid has boundary points in dimension i and -1 if not), which is approximately $2^{|\vec{l}|_1}$.

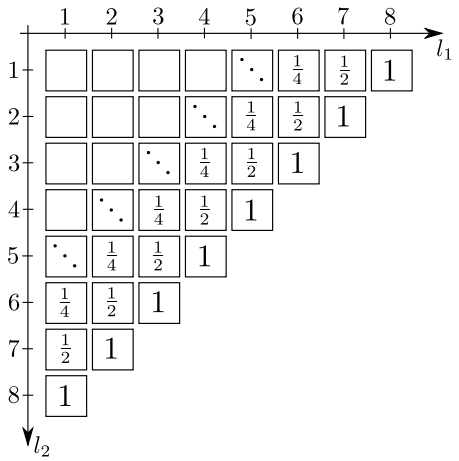

Figure 3.4.: Costs of component grids of different diagonals relative to the largest grids as estimated by the *linear cost model*.

It can be observed that all the component grids on a diagonal of the combination scheme have (nearly) the same number of grid points and that the number of grid points increases exponentially with the parameter a (see Equation (2.35)). This is visualized in Figure 3.4. Note that this Figure is a strong simplification. Especially when boundary points are present, even inside a diagonal there is a noticeable difference between the number of grid points of grids with isotropic discretization (same number of grid points in each dimension) and grids with very anisotropic discretizations. However, for many applications, the effects that influence the runtime are much more complex than it would be expressed by the number of grid points. If one would use the exact number of grid points one would create a sorting of the grids within a diagonal that might not at all be related to actual runtimes. The model would then falsely pretend a much higher accuracy than what it is actually capable of. For this reason, the model is used in the simplified form $2^{|\vec{l}|_1}$ throughout this work. It will be called the *linear cost model* in the following. It has also been used in the simplified form in [36].

Figure 3.5 shows the result of the scheduling algorithm using the linear cost model. In this idealized form, the scheduling algorithms finds the optimal solution to the bin packing problem. Here, for the optimal number of containers (larger than one) the difference in

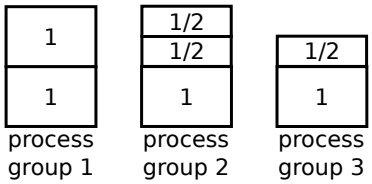

Figure 3.5.: Component grids of a two-dimensional combination technique distributed onto three process groups. The numbers indicate the relative costs determined by using the *linear cost model*.

run time of two process groups can be at most 1/2 of the time of the largest component grids in 2D [36] or $2^{-(d-1)}$ (size of the smallest grids in the combination scheme relative to the largest grids) this time for arbitrary dimensionality. The exponential decrease of the size of the component grids is highly beneficial to achieve good results with this scheduling algorithm. In the worst case where one process group is used for each component grid, the difference in runtime of the fastest and the slowest group will grow to $1 - 2^{-(d-1)}$ the time of the largest component grids.

The linear cost model is a reasonable metric for theoretical considerations and whenever no better model is available. However, it is in many cases insufficient to explain the relation between discretization and run time for actual implementations of numerical algorithms. In the context of the combination technique, an important factor that was identified to have a strong influence on the run time of a component grid is the anisotropy of the discretization. It has been observed that the runtime of the component grids on the same diagonal can differ significantly. For example, in the experiments with GENE described below, the most expensive component grid took almost three times longer to compute than the cheapest one in the same diagonal. The anisotropy of the discretization has a much stronger influence on the runtime than it could be explained by the number of grid points of the component grids.

The reasons for this behavior are manifold, and tailoring an exact cost model for a specific application requires a deep understanding of the numerical methods and computational algorithms that are used. A numerical reason to explain such a behavior could, for example, be the time step limit of explicit solvers, e.g., the well-known CFL condition. An unfavorably chosen anisotropic grid could result in a smaller time step size than an isotropic grid with the same number of grid points. Furthermore, an important factor to influence the runtime of a computational algorithm is the structure of the memory accesses. Due to the arrangement of the data in the memory, there are coordinate directions where a high number of grid points is less desirable than for other directions. Similar can be observed with distributed memory parallelization when an additional parameter is the number of processes in each direction. For GENE even different discretization methods are used in the different coordinate directions and thus the impact of the anisotropy is non-symmetric with respect to different coordinate directions.

In practice, creating a cost model that incorporates all the numerical and computational

details is usually out of scope for a sophisticated application code. Therefore, in this work a general cost model for the combination technique is proposed that builds on heuristics instead of application specific details and thus can be easily applied to a variety of applications. The underlying idea of this approach is to model the dependency of the runtime on the number of grids points and on the anisotropy of the discretization independently of each other. The model was developed within the EXAHD project in close collaboration with Christoph Kowitz and published in [49].

The two input variables of the model are the number of grid points

$$N(\Omega_{\vec{l}}) := 2^{|\vec{l}|_1} \tag{3.2}$$

and the anisotropy of the discretization expressed by the vector $\vec{s}_{\vec{l}} \in \mathbb{R}^d$, where

$$s_{\vec{l},i}(\Omega_{\vec{l}}) := \frac{l_i}{|\vec{l}|_1}. \tag{3.3}$$

It holds $|\vec{s}_{\vec{l}}|_1 = 1$ and $s_{\vec{l},i} = \frac{1}{d}$ for perfectly isotropic grids. A higher value than $\frac{1}{d}$ in one dimension results in lower values in at least one of the other dimensions and indicates an anisotropic discretization.

The proposed cost model has the form

$$t(\Omega_{\vec{l}}) = r(N)h(\vec{s}_{\vec{l}}), \tag{3.4}$$

where $r(N)$ models the relation between the application's run time and the number of grid points and $h(\vec{s}_{\vec{l}})$ scales this value according to the anisotropy of the discretization. In the following this model will be called *anisotropic cost model*.

The application's run time is modeled as

$$r(N) := mN^k + c, \tag{3.5}$$

with coefficients m, k and c. These coefficients are determined by fitting the model to measurements of the run time of a few reference grids (e.g., by the method of least squares). A reasonable choice for the set of reference grids is to use the (most) isotropic grid in each diagonal a. As there is not a perfectly isotropic grid in each diagonal, one could choose the grid with $\vec{s}_{\vec{l}}$ closest to the isotropic case ($s_{\vec{l},i} = \frac{1}{d}$) (e.g., with respect to the l_2-Norm) for these diagonals. This is visualized in Figure 3.6.

The run time of an application code does not necessarily grow linearly with the number of grid points for both, numerical and computational reasons. Depending on the choice of k, the term N^k can model an exponential increase as well. The constant c models the part of the run time which is independent of N. This constant has been observed to be especially important for small grids, where the constant part often dominated the run time.

The second term of the anisotropic cost model is independent of the number of grid points. It expresses the run time of a grid relative to the run time of the (isotropic) reference grids

$$\tilde{t}(\Omega_{\vec{l}}) \approx \frac{t(\Omega_{\vec{l}})}{r(N)}. \tag{3.6}$$

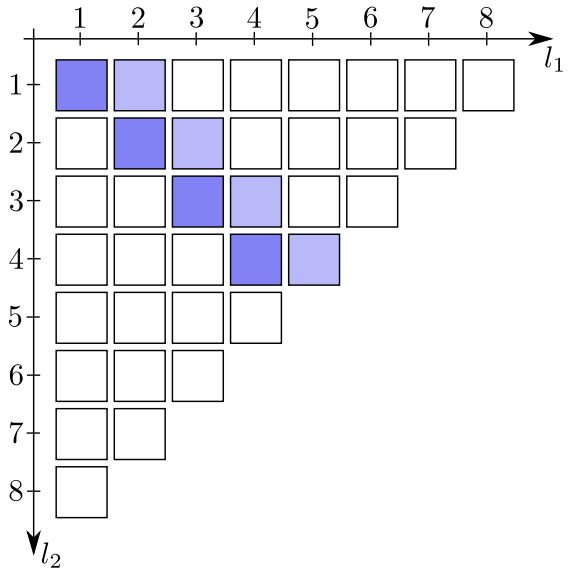

Figure 3.6.: One possible way to choose a reference grid with (nearly) isotropic discretization on each diagonal.

A polynomial ansatz for $h(\vec{s_{\vec{l}}})$ of the form

$$h(\vec{s_{\vec{l}}}) = c + \sum_{i=1}^{d-1} c_i s_{\vec{l},i} + \sum_{i=1}^{d-1} \sum_{j=1}^{i} c_{ij} s_{\vec{l},i} s_{\vec{l},j} + \sum_{i}^{d-1} \sum_{j=1}^{i} \sum_{k=1}^{j} c_{ijk} s_{\vec{l},i} s_{\vec{l},j} s_{\vec{l},k} + \cdots . \tag{3.7}$$

is used. As it holds $|\vec{s_{\vec{l}}}|_1 = 1$, only $d-1$ components of the d-dimensional $\vec{s_{\vec{l}}}$ can be chosen freely. Thus, the space of the polynomial reduces to $d-1$ dimensions. Polynomials of degree $p = 1, 2, 3$ have been used for $h(\vec{s_{\vec{l}}})$ in this work, which refers to cutting (3.7) after the second, third or fourth term respectively. The coefficients of $h(\vec{s_{\vec{l}}})$ are determined by fitting the model to the normalized measurement data $\tilde{t}(\Omega_{\vec{l}})$.

3.2.2. Accuracy of the anisotropic cost model

The accuracy of the *anisotropic cost model* and its effect on the load balance has been investigated for the application code GENE (see Chapter 5). The experiments were performed with the test case `standard/parameters_1` of GENE's test suite, the linear local initial value simulation of a single ITG instability. The run time was measured with different discretizations in the directions x, z, $v_\parallel$ and μ. In consistence with the experiments in Chapter 5, only one grid point was used in y-direction.

The measurements were performed with 32 MPI processes on a node of the supercomputer Hermit (see Appendix A). A minimal discretization level $\vec{l}_{\min} = (3, 1, 3, 3, 3)$ was used in order to fulfill the requirements of the simulation scenario. With a maximum discretization level $\vec{n} = (11, 1, 11, 11, 11)$ this resulted in a combination scheme with 425 component grids. As the goal of these experiments was solely to investigate the load

balancing, the reason behind the choice of this combination scheme was to produce a high number of component grids. The accuracy of the combined solution was not of interest in these experiments. Each component grid was run with an equal number of time steps. The time measured was the time for the actual initial value simulation. The time for the initialization routines, which can be relatively costly if only a small number of time steps is simulated, is not of interest for the cost model. These routines are only executed once in the beginning and can be neglected for production runs with several ten or hundred thousand time steps.

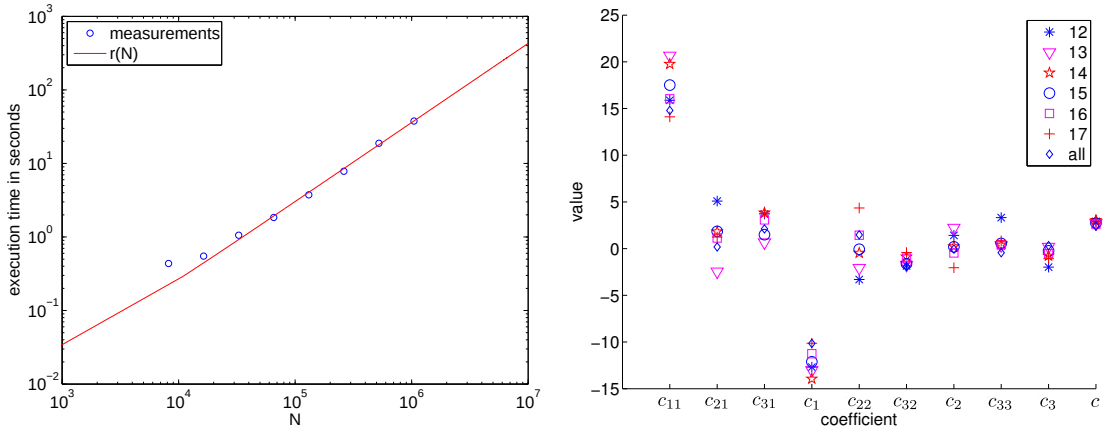

Figure 3.7.: **Left:** Function $r(N)$ for the run times of the (nearly) isotropic grids. **Right:** The coefficients of $h(\vec{s}_{\vec{l}})$ for different sets of training data. Reprinted from [49] with permission from IOS press.

Both terms of the cost model $r(N)$ and $h(\vec{s}_{\vec{l}})$ were fit individually to the measurement data in order to determine the model parameters. The model $r(N)$ was fit in the least-squares sense to the run times of the eight (nearly) isotropic grids of the diagonals $10 \leq a \leq 17$ with Matlab's `fit` function. This resulted in the parameters $m = 1.252 \cdot 10^5$, $k = 1.076$ and $c = 0.01308$. The result is visualized in Figure 3.7 (left). The model $r(N)$ covers the run time of grids with large N very well. Good accuracy for large grids is preferable over good accuracy for small grids, because a bad estimation of the run time of the largest grids results in a high absolute error of the cost model, whereas the impact of bad estimations of small grids has almost no influence.

Table 3.1.: Number of component grids for the different diagonals in the combination scheme with $\vec{l}_{\min} = (3, 1, 3, 3, 3)$ and $\vec{n} = (11, 1, 11, 11, 11)$.

a	14	15	16	17	total
	56	84	120	165	425

The following experiment investigates the assumption that the dependency of the run

time on the anisotropy is decoupled from the number of grid points. The parameters of $h(\vec{s_{\vec{l}}})$ were fitted by using only the measurements of the component grids in the combination scheme as training data that were on a certain diagonal a. The unofficial Matlab toolbox `polyfitn` [23] was used for this purpose. The function $r(N)$ was used with the parameters presented above to normalize the run times according to Equation (3.6). The set of test data to evaluate the accuracy of the model $t(\Omega_{\vec{l}})$ contained all the 425 grids in the combination scheme (the grids contained in this scheme are in the diagonals $14 \leq a \leq 17$). This procedure was repeated for different diagonals $14 \leq a \leq 17$ in order to investigate the impact of different training data on the accuracy of the model. Table 3.1 lists the different numbers of training data. Furthermore, the influence of different degrees p for the polynomial $h(\vec{s_{\vec{l}}})$ was investigated.

Table 3.2.: Mean value e_μ, standard deviation e_σ and root mean square e_{rms} of the relative error e of $t(\Omega_{\vec{l}})$ for different degrees p of the polynomial $h(\vec{s_{\vec{l}}})$. Each row in the table refers to a different set of training data.

a	e_μ			e_σ			e_{rms}		
	$p=1$	$p=2$	$p=3$	$p=1$	$p=2$	$p=3$	$p=1$	$p=2$	$p=3$
14	0.0392	0.0582	0.0592	0.1533	0.0838	0.0945	0.1581	0.1019	0.1114
15	0.0229	0.0220	0.0207	0.1491	0.0793	0.0751	0.1507	0.0822	0.0778
16	-0.0027	-0.0212	-0.0222	0.1437	0.0732	0.0663	0.1435	0.0762	0.0699
17	0.0605	0.0253	0.0292	0.1546	0.0828	0.0784	0.1658	0.0864	0.0836
all	0.0326	0.0163	0.0151	0.1487	0.0763	0.0691	0.1521	0.0779	0.0707

Table 3.2 shows the results. The relative error is defined as

$$e(\Omega_{\vec{l}}) := \frac{t(\Omega_{\vec{l}}) - t^*(\Omega_{\vec{l}})}{t^*(\Omega_{\vec{l}})},$$

where $t^*(\Omega_{\vec{l}})$ is the actual measurement of the run time and $t(\Omega_{\vec{l}})$ the estimation of the model. The root mean square error e_{rms} is defined as

$$e_{rms} := \sqrt{\frac{1}{n}\sum_{i}^{n} e_i^2}.$$

In the case *all* the training data consisted of all 425 component grids, i.e., the test and training data were identical. Comparing e_σ and e_{rms} for the same degree of the polynomial p, one can observe that these values vary only slightly for different training data, especially if the case $a = 14$ is ignored. Thus, the model $h(\vec{s_{\vec{l}}})$ produces a similar accuracy for the different sets of training data. This means, the effect of the anisotropy on the runtime of the component grids in one diagonal is representative for all other diagonals. This observation supports the claim that the influence of the anisotropy on the runtime of a component grid is decoupled from the actual size of the grid.

The case $a = 14$ had the lowest number of grids. As the anisotropy gets more extreme with increasing a, the extreme cases of higher diagonals are only poorly predicted by a

model with this training data. This explains that the case $a = 14$ was significantly worse than the other cases.

The mean value of the error e_μ shows similar trends as e_σ and e_{rms}, though being more erratic. However, the mean value of the error is less important as indicator for the quality of the cost model than the standard deviation and root mean square. A constant offset of the predicted run times does not matter for the result of the scheduling algorithm, because it would not change the order in which the grids are assigned.

Choosing a polynomial of degree $p = 2$ significantly improved the accuracy compared to $p = 1$. Going from $p = 2$ to $p = 3$ only a slight improvement can be observed in most cases. In case of $a = 14$ it even slightly decreased the accuracy. Thus, a reasonable choice for this application scenario in GENE would be $p = 2$. Using a higher degree of the polynomial than necessary would only increase risk of overfitting.

Figure 3.7 (right) shows the coefficients of $h(\vec{s}_{\vec{l}})$ with $p = 2$ for different sets of training data. Here, also the diagonals $a = 12, 13$ are included (but as in the other cases only those grids with $\vec{l} \geq \vec{l}_{\min} = (3, 1, 3, 3, 3)$). The formation of clusters indicates that the polynomials are similar for the different sets of training data. This further supports the assumption that the influence of the anisotropy on the runtime is similar on different diagonals.

3.2.3. Scheduling experiments

In the following, the impact of the linear cost model and the anisotropic cost model on the quality of the scheduling are compared. The same test case and combination scheme as in the previous section was used. Hence, 425 component grids were distributed onto different numbers of process groups. Here, a process group corresponded to one node on Hermit. The component grids were not actually recomputed for each number of process groups, as this is not necessary and would be a waste of computational resources. Instead the scheduling was simulated by using the actual run times for the component grids measured on a single node. It was made sure with a few sample runs that the simulations agreed with the actual results for multiple process groups.

With the anisotropic cost model the experiment was straight-forward. The list of component grids was sorted according to the costs model and the grids were distributed by the scheduling algorithm as described above. For the linear cost model the simplification $t(\Omega_{\vec{l}}) = 2^{|\vec{l}|_1}$ was used. In fact, here this is not a simplification, because powers of two are actually used for the experiments with GENE (see Chapter 5). A problem of this model for the scheduling algorithm is that there are many grids with exactly the same costs. This leads to the undesirable effect that the result of the scheduling algorithm strongly depends on the order in which the initial list (before it is sorted) of component grids is created. In order to to obtain statistically meaningful results, the experiments with the linear cost model were repeated 1000 times. Each time the list of component grids was randomly permuted before it was sorted.

Figure 3.8 shows the parallel efficiency E_P over the number of process groups P of the scheduling algorithm with the two different cost models. Here, the parallel efficiency is

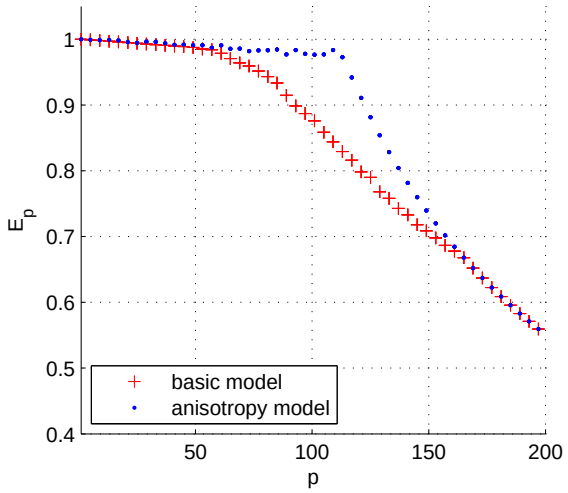

Figure 3.8.: Parallel efficiency E_P over the number of process groups P.

defined as

$$E_P = \frac{T_1}{PT_P}, \tag{3.8}$$

where T_1 is the time it takes to compute all component grids with only one process group and T_P the time with P process groups, respectively. For the experiments with the linear cost model the average of the 1000 runs is plotted. With the anisotropic cost model the efficiency stayed above 97% up to 113 process groups, before it drops rapidly. With the linear cost model an efficiency comparable to the anisotropic model was only observed until 50 process groups. The largest difference in efficiency between the two models was obtained with 110 process groups, where it was 14% higher with the anisotropic model. The two curves converge again with increasing number of process groups. They are (nearly) identical for $P \geq 165$. At this point the number of process group is identical to the number of the largest component grids (compare Table 3.1). With increasing P the impact of the cost model on the assignment of the component grids to the process groups gets smaller and smaller and the scheduling algorithm produces very similar results. After some point the overall runtime is (almost) completely dominated by the runtime of the most expensive component grid and the efficiency decreases (almost) linearly until there is only one grid per process group (of course, it would decrease even further if one would use more process groups than component grids). This experiments confirms the assumption that led to the manger-worker parallelization scheme: using one process group for each component grid result in severe load imbalances and thus a poor parallel efficiency for the coarse parallelization level.

One challenge of the anisotropic cost model for practical purposes is that the data to fit the model is usually not available a priori. However, if there is a series of simulation runs with only slightly varying application parameters (e.g., parameter or convergence studies), a reasonable strategy would be to use the first simulation run (e.g., with the linear cost model) to collect this data in order to create the cost model for the subsequent

runs. One idea for future work could be investigate strategies to automatically build the cost model on the fly, while running the simulation. After a component grid was computed for the first outer iteration, the runtime is known. So, at the beginning of a simulation after the first batch of component grids was computed, this data can already be used to create a coarse cost model. This model is then used to schedule the remaining component grids and can be refined incrementally whenever new data is available.

3.2.4. Rebalancing after the combination step

For the solution of time-dependent PDEs, where multiple outer iterations are computed, the load balancing can be further improved. After the first outer iteration the actual runtimes of the individual component grids are known and potentially a better distribution of the component grids to the process groups can be found than it was possible with the initial predictions of the cost model. In the following, a suitable rebalancing strategy is proposed. The rebalancing is performed either directly before or directly after the combination step, before the next outer iteration.

Algorithm 4: Rebalancing.

```
1  k ← 0
2  g_avg ← Σ_i^n g_i/n ;                    // compute average runtime of all groups
3  while k < k_max do
4      g_l ← max_i{g_i} ;                    // find runtime of slowest group
       // abort if load imbalance below threshold
5      if g_l/g_avg < ε then
6          exit
7      g_s ← min_i{g_i} ;                    // find runtime of fastest group
       // find task with runtime t_j on slowest group that would
          increase the time of the fastest group closest to g_avg
8      j = argmin_i{|g_s + t_i − g_avg|}
       // make sure that load imbalance actually decreases before
          moving the task
9      if g_s + t_j < g_l then
10         move task j to slowest group
11         k ← k + 1
12     else
13         exit
```

Perfect load balance is achieved when the runtime of each process group is identical to the average runtime of all process groups. Thus, load imbalance can be measured as the ratio of the slowest group to the average runtime. The idea of Algorithm 4 is to find the component grid (called *task* in Algorithm 4) on the slowest process group which,

when moved to the fastest group, increases the runtime of the fastest group closest to the average runtime. If the new runtime of the fastest process group is then smaller than the original runtime of the slowest group, the task is moved. Every iteration of this procedure decreases the load imbalance until it is not possible any more to find a task to be moved or if an appropriate abortion criterion is met. In order to avoid too many iterations, the algorithm can be aborted as soon as the load imbalance falls below a given threshold or if a maximum number of iterations is reached. Although, as for the bin-packing algorithm, this algorithm will not necessarily find the optimal solution, it is a simple and effective approach to rebalance the load of the process groups.

Moving the component grids is the most complex and expensive operation of this rebalancing strategy. When a component grid is created on a different process group, the corresponding initialization routine of the application code is run again. This can be an expensive operation (e.g., see Chapter 4). Another issue is to provide the data for the freshly created component grid, i.e., the combined solution projected into the function space of this grid. Essentially, there are two solutions to this issue. The straight-forward approach is to simply send this data from the old group to the new group. Another approach is to do this in the course of the combination step. If the strategy *Sparse Grid Reduce* (see Section 2.7) is used for the combination step, this is rather simple. In this case, one would create and initialize the component grid on the new group before the combination step. Then, after the combination step, the combined solution is available on the freshly created component grid and grid can be deleted on the old group. Note that this only works if the grid is none of the component grids with the highest level-sum, because the largest hierarchical subspaces usually are not transferred during the combination step. This could be solved by performing *Sparse Grid Reduce* with the full sparse grid space for a single time. However, as it is rather unlikely that one of the largest component grids has to be moved (this would mean that the initial distribution of the component grids was completely disadvantageous), this is not a real problem in practice. If the algorithm *Subspace Reduce* is used for the combination step, the combination algorithm must be slightly modified. When creating the set of communication partners for each hierarchical subspace, the destination process group of the component grid to be moved must be added as communication partner to the sets of the respective hierarchical subspaces.

The underlying assumption for the success of the rebalancing strategy is that the runtime of the component grids stays relatively constant for the remaining outer iterations. Although this is a reasonable assumption when a fixed number of inner time steps is used, with adaptive time-stepping this is not necessarily fulfilled. Furthermore, there might be issues related to the HPC system that lead to fluctuating runtimes of the component grids over time. In order to avoid rebalancing too frequently, in such cases it might be useful to track the runtime for multiple outer iterations and use statistical methods to decide whether rebalancing is actually necessary.

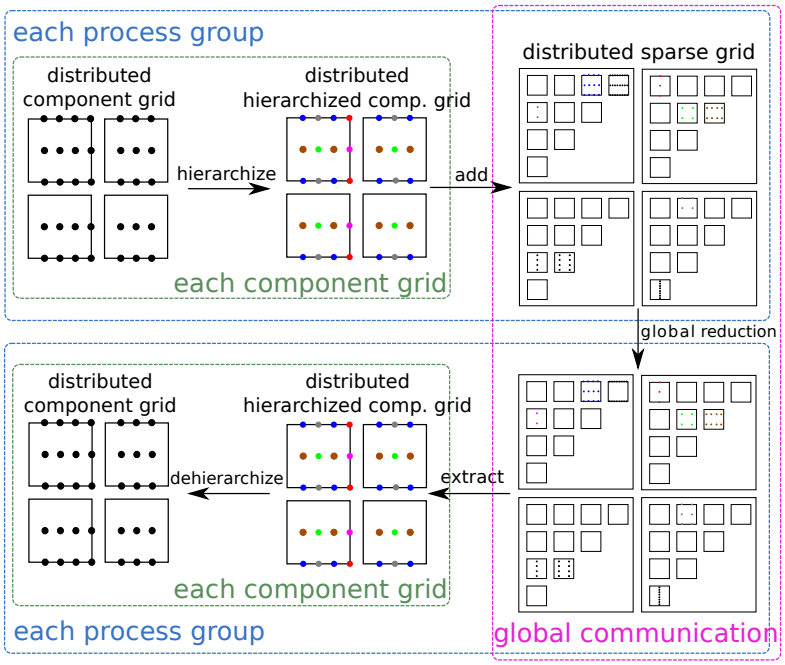

Figure 3.9.: The distributed combination step.

3.3. Efficient and scalable combination

The combination step is the only step in the parallel combination technique algorithm (compare Algorithm 3) that requires global communication between the process groups. Here, the term *global communication* refers to the communication between processes of different process groups, whereas *local communication* refers to communication between processes inside the same process group. The impact of the combination step on the overall performance gets more and more important, the shorter the combination interval is. In the worst-case scenario the component grids are combined after each time step. Thus, an efficient and scalable implementation of this step is crucial for the overall performance and the feasibility of the combination technique applied to time-dependent PDEs. One can speak of an efficient and scalable implementation, if the combination step does not consume more than a reasonable fraction of the overall run time of simulations with the combination technique in a massively parallel setup. In [50, 51], I presented new algorithms that enable this and demonstrated their scalability on up to $180,225$ cores of the supercomputer *Hazel Hen*. The following text is an extended version of these two publications.

In Section 2.7 two strategies for the combination step were proposed, *Sparse Grid Reduce* and *Subspace Reduce*. In the following, an implementation of *Sparse Grid Reduce* embedded in the structure of the massively parallel combination technique is presented. Although the main concept of *Sparse Grid Reduce* is maintained, the models presented in Section 2.7 are not directly applicable any more, because two underlying assumptions

are violated. First, there is no individual node (or process group) for each component grid, but instead multiple process groups are used, which compute and store multiple component grids each. The experiments in Section 3.2 indicate that in order to achieve a good load balance, the number of process groups should be significantly lower than the number of component grids. Second, not only a single node is used to compute each component grid. For the large-scale problems targeted in this work, the size of the component grids are expected to exceed the main memory of a single compute node and thus this is not feasible. Furthermore, even in the cases where it would be possible to store the grid, using only one compute node to process a component grid would severely limit the parallelization potential of the combination technique. Both assumptions are in favor for the superiority of *Subspace Reduce* for large problem sizes. However, the lower the number of process groups is and the higher the number of component grids handled by each group is, the less likely it is that a hierarchical subspace of the sparse grid does not exist in this process group. Thus, the potential of the communication structure of *Subspace Reduce* to save data transfers decreases with decreasing number of process groups, because there are less subspaces which are not exchanged between all groups. Additionally, the decreasing number of process groups is also beneficial for *Sparse Grid Reduce*, because of less communication partners for the *AllReduce* operation. Furthermore, with increasing size of the process groups, the data to be transferred by each node decreases. The investigations in Section 2.7 show that for small amounts of data, *Sparse Grid Reduce* is more favorable, because of the smaller number of individual messages to be sent. For these reasons, it is questionable whether *Subspace Reduce* can provide an outperformance over *Sparse Grid Reduce* which would justify the additional complexity of this algorithm.

The main substeps of the distributed combination step are depicted in Figure 3.9. These steps correspond to lines 7 to 16 in Algorithm 3. In each process group a distributed sparse grid data structure is created which will eventually contain the combined solution (line 8). First, all the component grids are hierarchized, one after the other (line 10), i.e. they are transferred from the nodal basis representation into the hierarchical basis of the sparse grid. Next, after being multiplied with the according hierarchical coefficient, each component grid is added to the sparse grid. Note that a component grid only contributes to a subset of the hierarchical subspaces in the sparse grid. For example, the component grid in Figure 3.9 has discretization level $\vec{l} = (3, 2)$. This means, it only contributes to the hierarchical subspaces $W_{\vec{l}'}$ with $\vec{l}' \leq (3, 2)$. The operation of multiplying the component grids with the combination coefficient and adding them up on the sparse grid data structure is called *local reduction* (line 11). After the component grids have been reduced locally inside the process groups, they are reduced globally across the process groups (line 12). Global reduction means to add up the individual sparse grids of the different process groups. Now, each process group holds the combined solution in the distributed sparse grid. The combined solution is then brought back onto the individual component grids in two additional local steps. During the so-called *scatter* operation, for each component grid the relevant hierarchical coefficients are extracted from the combined solution (line 15). The combined solution is now available on each component grid in the hierarchical basis representation. Finally, it is transferred back into the nodal

```
for dd ← 1 to d do
  for all 1-dim pole P in direction dd do
    for l ← l_dd to 1 do
      for all x_i on level l of pole P do
        x_i = x_i − 0.5 ∗ x_leftPredecessor
        x_i = x_i − 0.5 ∗ x_rightPredecessor
      end for
    end for
  end for
end for
```

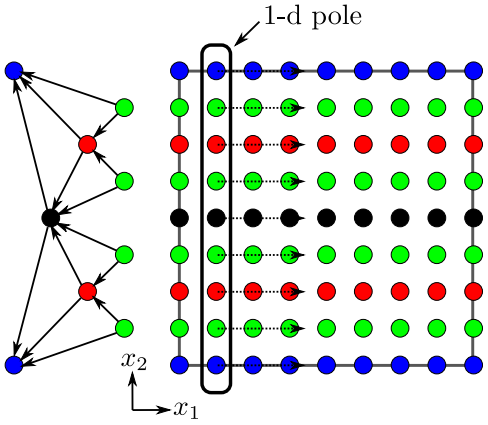

Figure 3.10.: **Left:** Unidirectional hierarchization principle (from [57]). **Right:** Hierarchization of the x_2-dimension. The graph shows the data dependencies for the hierarchization of each one-dimensional pole. Reprinted from [50] with permission from IOS press.

basis by dehierarchization (line 16). In the following, the individual substeps of the distributed combination step are investigated in detail.

3.3.1. Distributed hierarchization and dehierarchization

A very common approach for hierarchization and dehierarchization is the so-called unidirectional principle. Here, one exploits the underlying tensor structure and hierarchizes the d dimensions of the component grid one after each other. The following sections will only treat the hierarchization. However, everything said here is true for dehierarchization as well, as it is just the inverse operation. The mathematical background for hierarchization and dehierarchization is presented in Section 2.3.

Hupp presented a well optimized implementation of the unidirectional principle in [57]. Figure 3.10 (left) shows the basic algorithm. The algorithm traverses the grid in one-dimensional poles that are aligned in direction of the current dimension, performing one-dimensional hierarchization in each pole. As this algorithm works in-place, directly on the data structure of the grid, the coefficients x_i cannot be updated by simply iterating through the pole from beginning to end. Instead they have to be updated level-wise according to their discretization level, starting on the highest level. Figure 3.10 shows the dependency graph for the one-dimensional hierarchization. For all dimensions other than the innermost dimension, the grid points within a pole potentially are in different cache lines. In order to reuse the grid points of already loaded cache lines as much as possible, the poles traverse the grid in direction of the innermost dimension (except when the innermost dimension itself is hierarchized). This is indicated by the arrows in Figure 3.10.

Assuming a grid with a discretization level of 10 in one dimension and boundary

points, there would be 1025 grid points in each pole directed in this dimension. For double precision and the typical cache line size of 64 Bytes this would require only around 64 KBytes to store the whole pole inside the cache. Thus, even for significantly higher numbers of grid points a modern processor should be able to hierarchize the pole completely inside the cache. However, note that for the high-dimensional applications addressed in this work it is not very likely that the component grids have much more than discretization level 10 in one dimension.

For grids that are significantly larger than the cache size the algorithm is memory bound. For this case a lower bound for the runtime of the unidirectional principle is presented in [57]

$$2d \cdot (\#\text{gridpoints}) \cdot (\text{size of datatype})/(\text{memory bandwidth}).$$

Independent of the actual implementation of the unidirectional principle each grid point is loaded and stored d times. In [57], shared-memory parallelization (OpenMP), vectorization and blocking techniques are applied and a performance close to this bound is achieved.

Hupp presents a cache-oblivious hierarchization algorithm in [56] which avoids to iterate d times over the whole component grid as in the unidirectional principle. The algorithm recursively divides the component grid into smaller subproblems that completely fit into the cache. Each of the subproblem is then hierarchized via the unidirectional principle. This approach was faster than his implementation of the unidirectional principle in almost all cases.

These two approaches are limited to shared memory parallelism. However, the large-scale parallel applications addressed in this work, typically use MPI parallelization and computational grids that are distributed onto a large numbers of compute nodes of an HPC system. In order to efficiently hierarchize such distributed component grids, I developed a new hierarchization algorithm based on the unidirectional principle [50].

Unlike for the approaches presented above, where the whole component grid is available in the main memory of a node and can be directly accessed by all its processors, this is not possible when the component grid is distributed over multiple nodes. The data stored on a remote node cannot be directly accessed, but instead has to be transferred over the network using an interface such as MPI. Figure 3.11 illustrates this with a component grid distributed onto four different MPI processes. The dependency graph for the hierarchization of the x_2-direction visualizes that data from the subgrids of other process is required in order to update the local grid points.

Thus, the new distributed hierarchization algorithm has an additional communication step. The new algorithm can roughly be sketched as:
For each dimension

1. Calculate the dependency graph for the current dimension and exchange the values via MPI.

2. Hierarchize the local subgrid in direction of the current dimension.

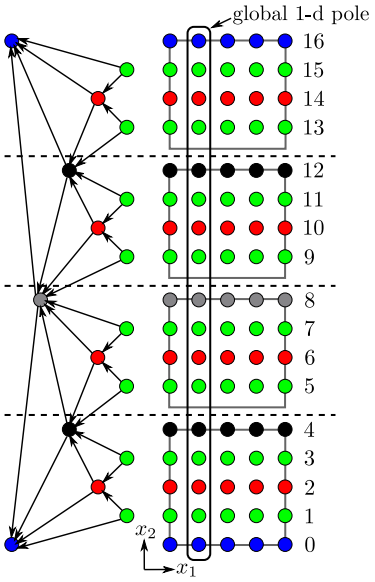

Figure 3.11.: Dependency graph for the hierarchization the x_2-direction for a component grid distributed onto four processes. Reprinted from [50] with permission from IOS press.

In order to determine which grid points need to be exchanged, each process sets up the dependency graph. As the graph can be deduced from the discretization and the domain partitioning, which is known to all processes, this does not need any further communication. Each node in the dependency graph corresponds to a set of grid points, a $(d-1)$-dimensional slice of the grid. For the hierarchization, each process checks for each node in the dependency graph, whether the direct hierarchical predecessors are contained in the local subdomain of the process, or whether they are contained in the subdomain of a remote process. For each hierarchical predecessor that is contained on a remote process, the local process will receive and store the $(d-1)$-dimensional grid of data, so-called *remote data*. In the same way, each process also determines which data it needs to send to the other processes. The data exchange is implemented by non-blocking MPI calls. In the case of dehierarchization, it is not enough to exchange only the direct hierarchical predecessors, but instead one needs to exchange the whole hierarchy of predecessors (all predecessors of the hierarchical predecessor). For the example in Figure 3.11 the hierarchical predecessors of grid point number 4 are the points 0 and 8, and those of the point 8 are 0 and 16. The reason is that the dehierarchization algorithm requires that the grid points of the previous level are already dehierarchized before the next level can be processed. This can either be solved by exchanging the direct hierarchical predecessors after each level, or by exchanging the whole hierarchy once in the beginning. However, exchanging data after each level results in significantly worse performance due to the additional synchronization points.

After the data has been exchanged, each process locally performs the hierarchization

very similar to Hupp's algorithm: The local subgrid is traversed with 1-dimensional poles. However, for the hierarchization of a pole it can occur that a hierarchical predecessor is not contained in the local domain, but instead it has to be read from the remote data. Deciding for the update of each grid point whether the hierarchical predecessor is contained in the local subgrid or in the remote data, requires additional logic and address calculations. This resulted in very bad performance. In order to circumvent this problem, the data of the current pole (local and remote) is copied to a temporary array which has the full global extent of the dimension being currently hierarchized. Then the current pole can be hierarchized in the temporary array using an efficient 1-d hierarchization kernel. Afterwards, the updated values are copied back from the temporary array to the local subgrid. The temporary array fits easily into the cache, even for very high discretization levels. As with Hupp's algorithm, the local subgrid as well as the data structures for the remote data are accessed in a cache-optimal order.

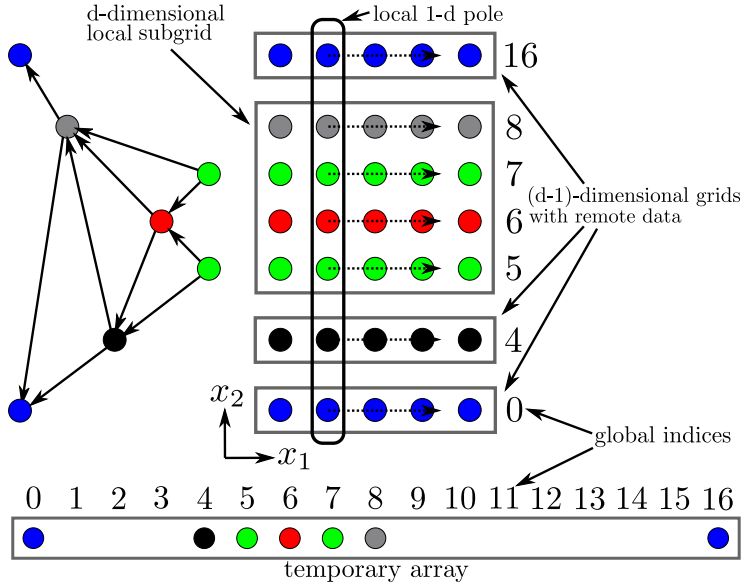

Figure 3.12.: Distributed hierarchization with remote data from the local view of one process. Reprinted from [50] with permission from IOS press.

Figure 3.12 illustrates the hierarchization of the local subgrid for the second process from the bottom in Figure 3.11. The d-dimensional local subgrid of this process holds the global indices 5 to 8 of the x_2 direction. Additionally, the process received the remote data with global indices 0, 4 and 16 in form of $(d-1)$-dimensional grids during the communication step. This is consistent with the dependency graph for the hierarchization of the local subgrid. Furthermore, it is visualized how the data in the pole maps to the temporary array.

3.3.2. Scalability of the distributed hierarchization

The scalability of the distributed hierarchization algorithm was investigated on Germany's supercomputer *Hornet* at HLRS, which is the predecessor of the current system *Hazel Hen* (see Appendix A). The architecture is identical to *Hazel Hen*, but the system consisted only of 3944 nodes.

The component grids used for the measurements are described by the dimensionality d, the discretization levels $\vec{l}$ and the number of processes $\vec{p}$ per dimension used for the domain decomposition. The entries in $\vec{p}$ were always powers of two. Although this is not a condition for the distributed hierarchization algorithm, it is a natural choice considering the number of grid points per dimension which (almost) are powers of two as well. Additionally, it simplified the setup of the experiments, as the total number of processes for every possible domain decomposition is a power of two as well. Strong and weak scaling experiments with 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 and 32768 cores on Hornet were performed using all 24 cores per node. Strong scaling means to increase the number of processors while the problem size stays constant. Thus, the number of grid points on each process decreases. Weak scaling means to increase the problem size proportionally to the number of processors. Thus, the number of grid points per process stays (roughly) constant. The grids had boundary points in all dimensions, thus representing the worst-case scenario in terms of runtime. Using boundary points does not only mean that there are more grid points to be processed, but also more data to be exchanged over the network during the communication step.

In general, the time per process to exchange the data as well as to hierarchize the local subgrid can be noticeably different on each process. There are various reasons that can explain this observation. First of all, there are algorithmic reasons: The processes have to compute (slightly) different numbers of grid points. Considering the communication structure, some processes also have to exchange more data than other processes. But the differences in the runtime of individual processes also have technical reasons. The effective network bandwidth that is achieved when transferring data between two particular nodes depends on the position of these nodes in the network topology. Furthermore, it is externally influenced by the overall load on the network and thus it can fluctuate over time. All the 24 processes on each node share the main memory bandwidth and the network bandwidth and so even within a node not every process might achieve the same effective bandwidth. Thus, the most meaningful result is the average time per process. Nevertheless, it was observed that the maximum runtime (of the slowest process) scales in the same way as the average time. Unless otherwise stated, the average runtime is presented in the following.

Figure 3.13 shows scaling results for grids of different dimensionalities. The grids that were used here had isotropic discretization and isotropic domain decomposition. A domain decomposition is considered to be anisotropic if the number of processes per dimension is not proportional to the number of grid points per dimension. Actually, neither the discretization nor the domain decomposition were completely isotropic in all cases. However, the components of $\vec{l}$ did not differ by more than one. Likewise, the components of $\vec{p}$ did not differ more than a factor of two. Every time when increasing

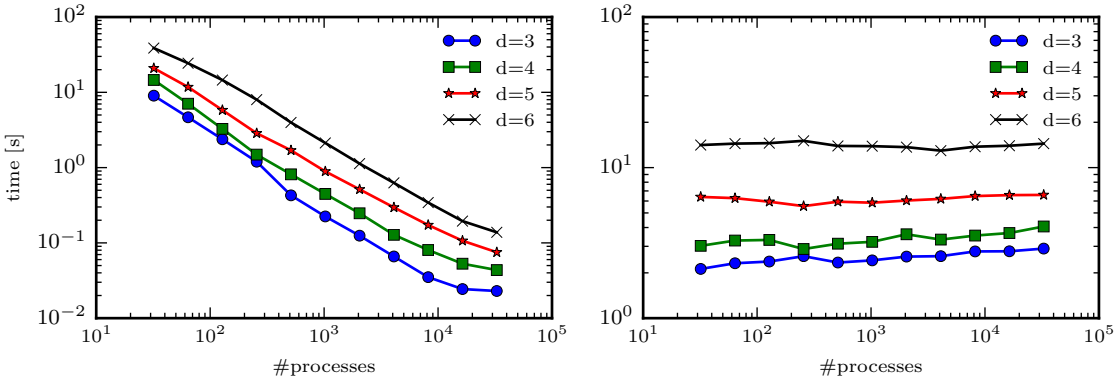

Figure 3.13.: Strong **(left)** and weak **(right)** scaling results for the hierarchization of grids of different dimensionalities. Reprinted from [50] with permission from IOS press.

the number of processes, the smallest component of $\vec{p}$ was doubled.

For strong scaling the grids were chosen so that they had a comparable size: $\vec{l} = (11, 11, 10)$ (34.4 GB), $\vec{l} = (8, 8, 8, 8)$ (34.9 GB), $\vec{l} = (7, 7, 6, 6, 6)$ (36.5 GB) and $\vec{l} = (6, 6, 5, 5, 5, 5)$ (40.08 GB). The runtime increases with dimensionality, because of the d stages of the unidirectional principle. However, the algorithm scales well independent of the dimensionality. For $d = 5$ a speedup of 278 was obtained with 32768 processes compared to the result on 32 processes. This is a good result if one considers that with 32768 cores the local subgrid of each process was only a few megabytes in size. The weak scaling experiments were started with grids of $\vec{l} = (10, 10, 10)$ (8.6 GB), $\vec{l} = (8, 8, 7, 7)$ (8.7 GB), $\vec{l} = (6, 6, 6, 6, 6)$ (9.2 GB) and $\vec{l} = (5, 5, 5, 5, 5, 5)$ (10.3 GB). The grid size was increased proportionally to the number of processes, by incrementing the smallest component in $\vec{l}$, each time. For $d = 5$, the runtime only grows from 6.38s (32 cores) to 6.58s (32768 cores). This corresponds to a parallel efficiency of almost 97%.

Figure 3.14 shows the individual times of the communication step and the hierarchization step for the case $d = 5$ of the experiment described above. The values are the aggregated times, summed over all directions in the algorithm. Additionally, the maximum time (of all processes) for the communication step is presented. For the hierarchization step the difference between the maximum time and the average time is hardly noticeable, so only the average time is presented. For strong scaling, the number of local grid points decreases and so does the time for the hierarchization step. In the same way the size of the blocks of data that need to be exchanged decreases. However, the amount of data exchanged differs significantly between the processes. Considering the communication structure (compare Figure 3.11), there are always processes that have to send or receive more data than others. This explains the significant difference between maximum and average communication time.

Figure 3.14 also shows the individual communication and hierarchization time of a naive implementation which is based on gathering the component grid on a single node. Here, a serial implementation of the unidirectional principle as presented in Figure 3.10

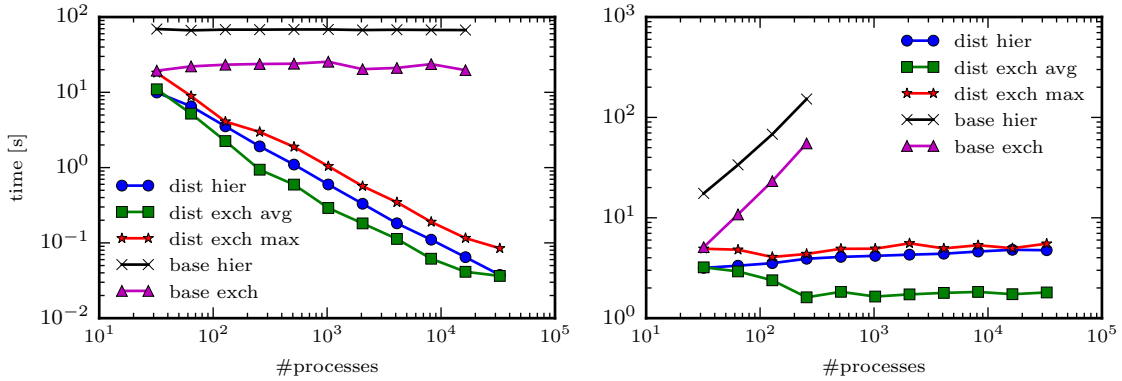

Figure 3.14.: Strong **(left)** and weak **(right)** scaling results for a 5-dimensional grid. Individual times for data exchange and hierarchization for the distributed hierarchization algorithm (dist) and for a straightforward implementation using serial hierarchization (base). Reprinted from [50] with permission from IOS press.

is used for the hierarchization. This test was included to demonstrate that such an approach cannot scale up to large numbers of processes. For the communication step, the limiting factor is the node's network bandwidth. Thus, the time to gather the grid remains constant for strong scaling and grows proportionally to the grid size for weak scaling. Obviously, the time for hierarchization cannot scale because the resources that can be used for the computations are limited to one node. Furthermore, this approach limits the size of the component grid to the size of the main memory of a single node (128 GB on Hornet).

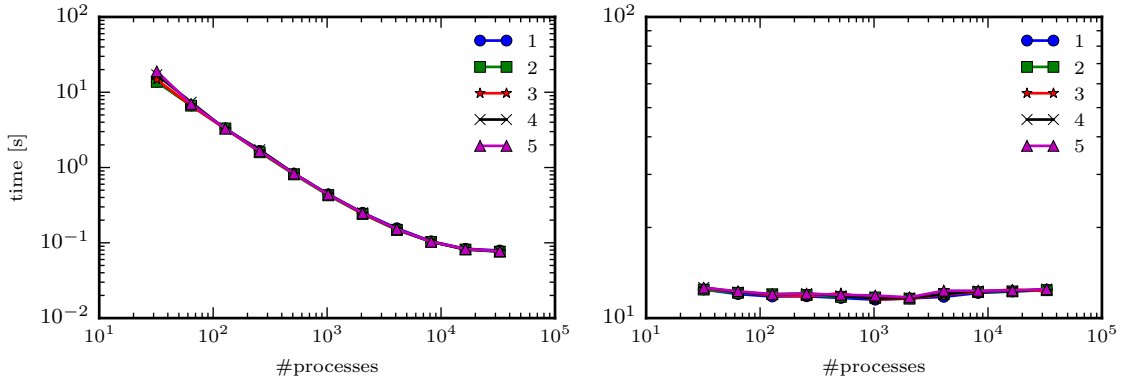

Figure 3.15.: Strong **(left)** and weak **(right)** scaling results for the distributed hierarchization of an anisotropic 5-dimensional grid. For each case the number denotes the anisotropic dimension. Reprinted from [50] with permission from IOS press.

The following experiment investigates how the anisotropy of the discretization influences

the scalability of the distributed hierarchization algorithm. Although, theoretically discretizations with level 1 in all but one dimension can occur for the combination technique, such extreme case are often excluded in practice, because of physical or numerical constraints. The test case used for strong scaling had discretization level 16 in one dimension and level 4 in all the other dimensions. Thus, the finest dimension had 4096 times more grid points than the others. The weak scaling experiments were started with an isotropic grid with $\vec{l} = 6, 6, 6, 6, 6$ on 32 processes. When the number of processes was increased, the discretization was only refined in one dimension. This procedure resulted in a discretization level of 16 for the finest dimension. In both cases, strong and weak scaling, the domain was only decomposed in direction of the finest dimension. Figure 3.15 shows that there was no noticeable difference which direction was refined. Although, this was expected for dimensions 2 to 5, it was not clear that there is no difference for the innermost dimension. The innermost dimension is aligned with the cache lines and has a different memory access pattern than the other dimensions. For strong scaling, the speedup with 32768 process (compared to 32 processes) was 176, which was about 62% of the speedup measured in the isotropic case. As in the isotropic case, the weak scaling was almost perfect. The runtime did even slightly decrease from 12.44s (32 processes) to 12.42s (32768 processes).

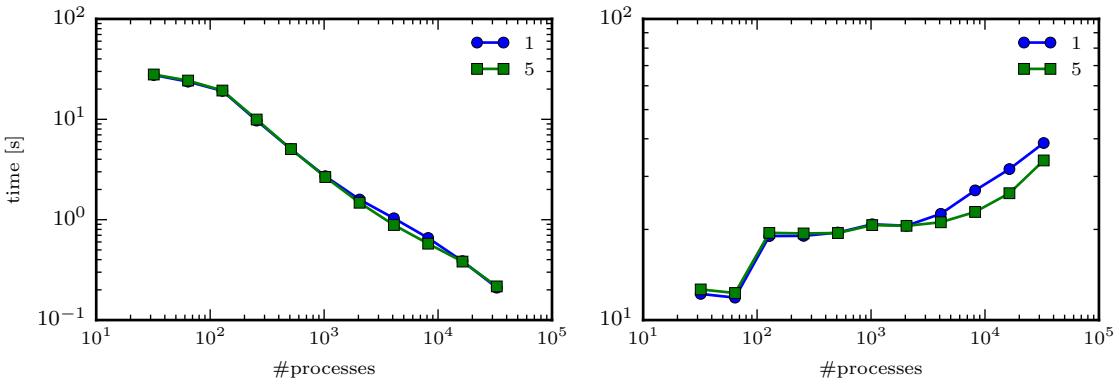

Figure 3.16.: Strong **(left)** and weak **(right)** scaling results for the distributed hierarchization of a 5-dimensional grid which has an anisotropic domain decomposition either in dimension 1 or dimension 5. Reprinted from [50] with permission from IOS press.

Figure 3.16 shows the the influence of extremely anisotropic domain decompositions on the runtime of the hierarchization algorithm. Here, for a grid with isotropic discretization, the number of processes was increased only in one dimension. Due to the symmetry in the above results, the experiments were only performed in the innermost dimension and an arbitrary other dimension (dimension 5). For strong scaling, the grids were $\vec{l} = (7, 7, 6, 6, 6)^T$ (36.6 GB) and $\vec{l} = (6, 6, 6, 7, 7)^T$ respectively. Initially the grid was decomposed with 32 processes in one dimension and the number of processes was only increased in this dimension in order to create an extreme anisotropy in the domain decomposition. However, as there must be at least one grid point per process in each

dimension, when this number was reached, the discretization level of the next dimension was increased. This explains the kink in the plot with 128 processes. The performance of the algorithm was significantly worse than in the experiments above which had isotropic domain decomposition. For isotropic domain decompositions the amount of data to be exchanged with other processes becomes smaller when more processes are used. This is not the case with very anisotropic domain decompositions. Here, the data to be exchanged remains almost constant.

For weak scaling, the experiment was started with $\vec{l} = (6,6,6,6,6)^T$ and always the smallest entry in $\vec{l}$ was increased. As before, the domain was only decomposed in the dimension of interest. There is no real increase in the anisotropy of the domain decomposition when going from 32 to 64 processes, because the discretization level in the same dimension is increased as well. Thus, anisotropy in the domain decomposition did not exist before 128 processes, explaining the jump in the plot. In the case of such extremely anisotropic domain decompositions weak scaling is not possible, because the data to be exchanged grows with the problem size. However, note that such extreme domain decompositions will seldom be chosen in practice, because it is very likely that the application code will not perform well with such a configuration as well.

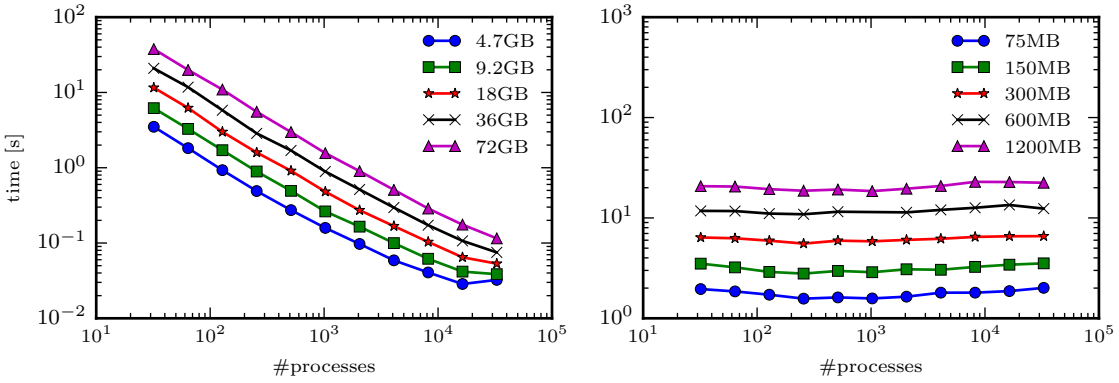

Figure 3.17.: Strong **(left)** and weak **(right)** scaling results for the distributed hierarchization of 5-dimensional grids with different grid sizes. For weak scaling the numbers denote the size of the local subgrid on each process which remains (roughly) constant. Reprinted from [50] with permission from IOS press.

In another experiment it was investigated how the actual size of the grids influences the scalability. Figure 3.17 shows the results. Here, discretization and domain decomposition of the grids were kept as isotropic as possible (increasing $\vec{l}$ and $\vec{p}$ where they have the lowest entry). For strong scaling, grids with $\vec{l} = (6,6,6,6,5)^T$ (4.7 GB), $\vec{l} = (6,6,6,6,6)^T$ (9.2 GB), $\vec{l} = (7,6,6,6,6)^T$ (18 GB), $\vec{l} = (7,7,6,6,6)^T$ (36 GB), $\vec{l} = (7,7,6,6,6)^T$ (72 GB) were used. For weak scaling, the experiment started with $\vec{l} = (6,6,6,5,5)^T$ (75 MB), $\vec{l} = (6,6,6,6,5)^T$ (150 MB), $\vec{l} = 6,6,6,6,6$ (300 MB), $\vec{l} = (7,6,6,6,6)^T$ (600 MB) and $\vec{l} = (7,6,6,6,6)^T$ (1200 MB) on $\vec{p} = (2,2,2,2,2)^T$ processes. The numbers in the brackets indicate the approximate size of the local subgrid on each process, which remains

(roughly) constant for different numbers of processes. Note that the largest grid on 32768 processes had more than $4 \cdot 10^{12}$ grid points and occupied almost 36 TByte of main memory. One can observe that in most cases the actual grid size did not have a noticeable impact on the performance of the hierarchization algorithm. Only in the case of the 4.7 GB grid the runtime could not be reduced further when going from 16384 to 32768 processes. For relatively small grids the communication does not scale any more. Even though in general the amount of data to be exchanged becomes smaller when more processes are used, the number of messages increases due to the increasing number of communication partners. Each message, however, has a constant overhead, independent of the message size (compare Section 2.7).

Figure 3.17 nicely demonstrates the five different grid sizes that occur in a five-dimensional combination technique. Usually the size of a process group would be chosen so that the best application performance will be achieved for the largest component grids, as their computation amounts for the largest fraction of the total runtime. Up to 16384 processes the hierarchization algorithm was able to efficiently compute even the smallest grids. Note that in the case of the 4.7 GB grid with 16384 processes the local subgrid stored by each process was only about 280 KB in size! However, even with 32768 processes the bad performance for the hierarchization of the smallest grids would be hardly noticeable in the absolute runtime of the hierarchization step. The number of small grids is rather low and so is the contribution of these grids to the total runtime of the hierarchization step.

The above experiments nicely demonstrate that the distributed hierarchization algorithm provides good performance for the extreme shapes of grids that occur for the combination technique. It enables an efficient and scalable treatment of highly anisotropic discretizations and domain decompositions, independent of the dimensionality or the size of the grid.

For dehierarchization the same trends were observed as for hierarchization. However, the absolute runtimes are a little higher and it scales slightly worse due to a more expensive communication step. As the whole hierarchy of hierarchical predecessors has to be exchanged, this results in more data to be transferred over the network, as well as in more communication partners.

Finally, the potential for further optimization of the distributed hierarchization algorithm will be discussed. Although the amount of data exchanged during the communication step cannot be further reduced, the communication algorithm allows for small improvements. Many processes have to send the same block of data to multiple recipients. This might be improved by using collective MPI calls, e.g., *MPI_Bcast*. Higher optimization potential is expected by hiding some of the computations of the hierarchization step behind data transfers. The grids containing the remote data would not be exchanged in one piece, but instead they would be exchanged in multiple smaller blocks. The blocks would be arranged so that after the first set of blocks was exchanged, the computations can already begin, while the next block is being exchanged.

Potentially the cache-oblivous hierarchization algorithm [56] could be extended to handle distributed component grids. Although the time for the computations can theoretically be reduced by a factor d, there is no real potential to reduce communication.

Independent of the order in which the grid points are updated, each grid point still has the same data dependencies as with the unidirectional principle. Thus, the same amount of data hast to be exchanged between the processes. As the communication consumes a considerable fraction of the runtime, one would not expect a huge performance benefit from a distributed cache-oblivious hierarchization algorithm.

3.3.3. Local reduction / scatter of component grids inside the process group

In the following, three different variants of the distributed sparse grid are presented. In general, the right choice of the data structure for the distributed sparse grid strongly depends on the domain decomposition of the application code. Note that only the reduction step is discussed here, as scattering the combined solution back onto the component grids is just the inverse operation.

Variant 1: Decomposition of the distributed sparse grid into hierarchical subspaces. One possibility for a distributed sparse grid data structure is to store each hierarchical subspace on exactly one process. A natural rule to assign the hierarchical subspaces to the processes would be to distribute them so that a balanced number of grid points is achieved. This is desirable, because the global reduction step is most efficient for an equal number of grid points on each process. Assigning the hierarchical subspaces to the processes is very similar to the bin packing approach used to distribute the component grids on the process groups (see Section 3.2). The subspaces are sorted by size in descending order and then they are assigned to the compute nodes in a round-robin fashion. In a second step, they are distributed to the processes of each node in the same way. It is more important to find a balanced distribution over the compute nodes than over the processes, because the two most critical resources for the reduction operation, network bandwidth and main memory bandwidth, are shared by all processes within a node.

The main advantage of this variant of distributed sparse grid is that it is completely independent of the discretization and parallelization of the component grids. Thus, with this variant, distributed component grids that have different domain decomposition can conveniently be added up. Furthermore, with this variant of the distributed sparse grid it is straight-forward to extend the global reduction step to process groups of different sizes. In order to add a component grid to the sparse grid, the hierarchical coefficients of the hierarchical subspaces that exist on both, the component grid and the sparse grid, are added. As illustrated in Figure 3.18, in the distributed component grid each hierarchical subspace is geometrically distributed over all processes of the process group. In order to add the coefficients of a hierarchical subspace to the corresponding subspace in the distributed sparse grid, the coefficients have to be gathered on the process which stores this subspace. The communication overhead of these gather operations is the major disadvantage of this distributed sparse grid variant.

The communication costs that incur when a distributed component grid is added to a distributed sparse are best understood if one considers the inverse operation, transferring the hierarchical coefficients from the sparse grid to the component grid. Each process stores N/m grid points of the distributed component grid (N grid points distributed

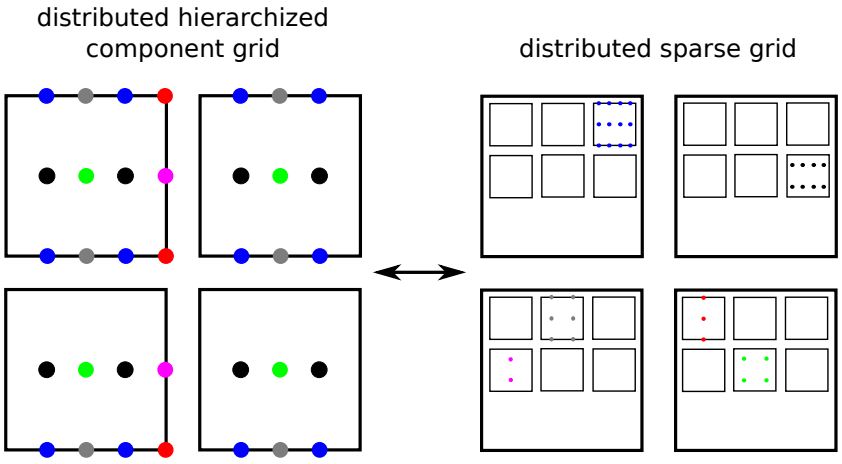

Figure 3.18.: Storage location of hierarchical coefficients. In the distributed component grid, the coefficients of a hierarchical subspace are distributed over all processes of the process group according to the geometrical domain decomposition. In the distributed sparse grid, a hierarchical subspace is stored by exactly one process of the process group. The colors indicate the coefficients that belong to the same hierarchical subspace.

over m processes), as well as N/m grid points of the distributed sparse grid. Due to the different assignment of grid points to the processes in the two data structures, each hierarchical subspace of the distributed sparse grid has to be scattered to $m-1$ other processes. Hence, each process sends $m-1$ messages of (approximately) N/m^2 grid points in size. This is visualized in Figure 3.19. In total each process has to send (and receive)

$$\frac{N}{m^2}(m-1) \approx \frac{N}{m} \tag{3.9}$$

grid points, which is approximately N/m for large m. Thus, the amount of data transferred by each process decreases linearly with the number of processes – a very desirable result, one would think. Unfortunately, this simple communication model does not consider that on actual systems the time to send a message to another process does not only depend on the amount of data to be transferred, but is bounded from below by a constant latency. Using the model introduced in Section 2.7, the time to send a message is $t = L + K/B$, with latency L, message size K and bandwidth B (unit: grid points per second). The total communication time is then

$$m(L + \frac{N}{Bm^2}) = mL + \frac{N}{Bm} \tag{3.10}$$

When m increases, the time for the actual data transfer decreases. Unfortunately, the constant overhead due to the message latency increases linearly with m and eventually dominates the communication time. Thus, scalability of this approach is severely limited.

The following experiment was performed to measure the time for the local reduction and scatter operation with this variant of distributed sparse grid. First, each process

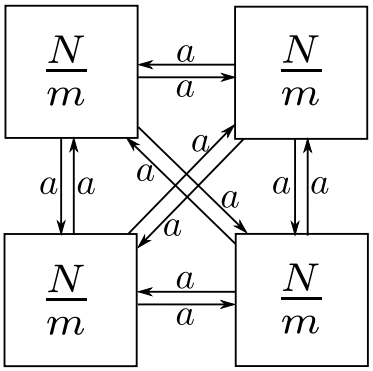

Figure 3.19.: Model for the communication when adding a distributed component grid to a distributed sparse grid of variant 1. Each process hold N/m grid points of the distributed component grid (N grid points distributed onto m processes) and of the distributed sparse grid. Each process sends $a = N/m^2$ grid points to $m - 1$ other processes.

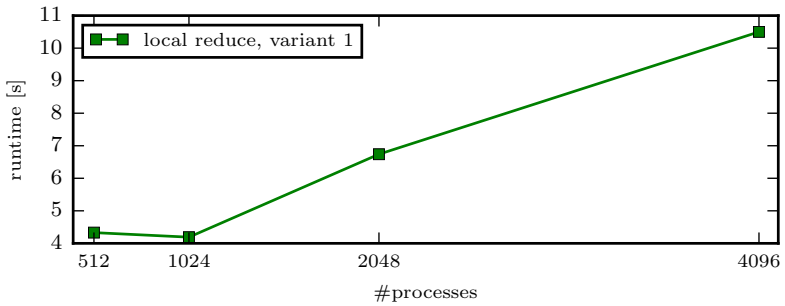

Figure 3.20.: Communication time to add a component grid with $\vec{l} = (10, 4, 6, 6, 6)$ to a distributed sparse grid of variant 1 for different numbers of processes.

creates a send buffer and a receive buffer for each other process. Then it determines for each grid point to which process it needs to be sent and copies the corresponding value into the respective buffer. Non-blocking `MPI_Isend` and `MPI_Irecv` calls are used to exchange the data. This enables the system to optimize the order of the messages, so that an optimal bandwidth usage can be achieved. After the data was exchanged, the coefficients are copied (or added) from the receive buffer to the right place in the underlying data structures. Figure 3.20 shows the time to exchange the data for a relatively large component grid with about 38 GB in size. The size for each message decreases rapidly from about 146 KB with 512 processes to only about 2 KB with 4096. A small decrease in run time from 512 to 1024 processes can be observed, before the runtime increases almost linearly. This supports the hypothesis derived from the communication model presented above. The number of messages can be reduced at the price of more communicated data volume by using collective operations like `MPI_Reduce`. However, a significant improvement in communication time could not be obtained with these

modifications.

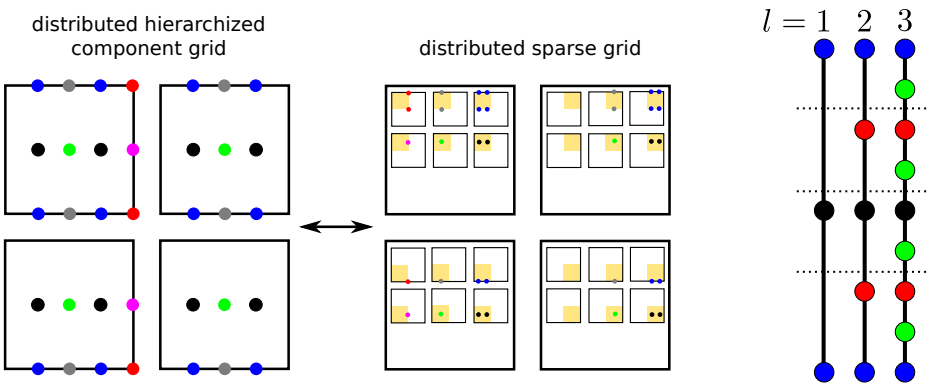

Figure 3.21.: **Left:** Each process stores for each hierarchical subspace only its local part of the domain. **Right:** One-dimensional component grids of different discretization levels, but equal geometric domain decomposition, distributed over four processes. Due to the nested structure of the grid, grid points with the same coordinates are always in the same subdomain, independent of the discretization level.

Variant 2: Communication-free local reduction of uniformly parallelized component grids. This variant of the distributed sparse grid considers the case that all component grids have the same geometric domain decomposition. It is then possible to perform the local reduction without any communication. In this variant, the hierarchical subspaces of the sparse grid are geometrically decomposed in the same way as the component grids and distributed over all processes of the process group. This is visualized in Figure 3.21 (left). Here, the assignment of a grid point to a process in the distributed sparse grid depends on its coordinate in the simulation domain. Due to the nested structure of the grids, the assignment of the grid points to the processes depends only on the decomposition of the domain, i.e., the chosen parallelization. Even if the component grids have different discretization, grid points with the same coordinates are always assigned to the same process (compare Figure 3.21 (right)). Thus, when the same geometric domain decomposition is used, the assignment of the grid points to the processes is equal for the distributed component grids and the distributed sparse grid.

The advantage over variant 1 is that no data has to be exchanged in order to sum up the hierarchical coefficients of the component grids (or transfer them back from the sparse grid). Here, adding a distributed component grid to the distributed sparse grid solely is a local operation: each process adds the hierarchical coefficients of its local part of the distributed component grid to its local part of the distributed sparse grid. Obviously, this operation scales perfectly with the number of processes. The main disadvantage of this approach is that it is limited to the case when all component grid are uniformly parallelized.

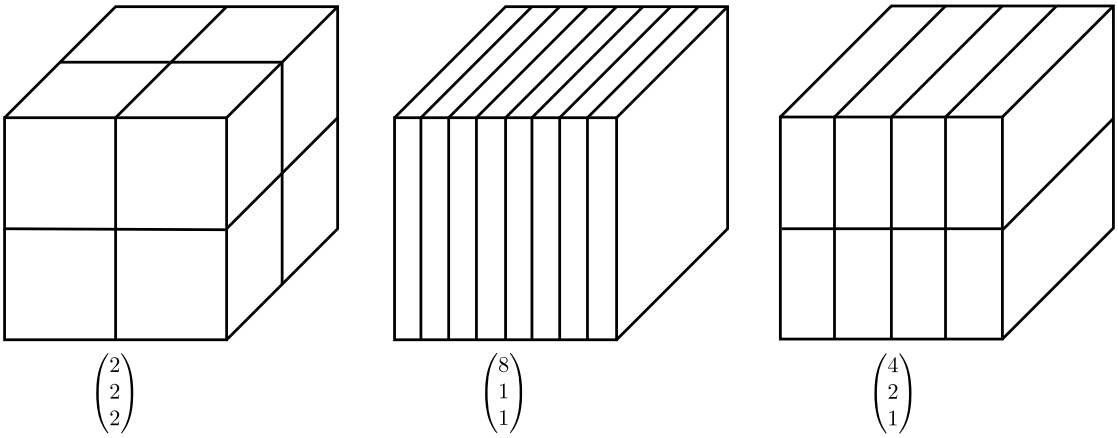

Figure 3.22.: Different geometric domain decompositions of a three-dimensional domain. The vectors indicate the number of processes in each dimension.

Variant 3: Geometric decomposition of the distributed sparse grid. In the third variant, the distributed sparse grid is geometrically decomposed. As in the second variant, each hierarchical subspace is distributed over all processes. However, unlike in the second variant, here the component grids are not uniformly decomposed. Although this variant has higher communication effort than the second variant, it allows for more flexibility by not enforcing the same domain decomposition for all component grids. Unlike in the first variant, this variant has a scalable communication structure. This will be investigated in the following.

Figure 3.22 shows an isotropic decomposition (left) and two anisotropic decompositions of a three-dimensional domain. Assume the left decomposition to correspond to the distributed sparse grid and the other two decompositions to correspond to component grids. Now, if one picks an arbitrary subdomain of one of the component grids and observes with which subdomains on the distributed sparse grid this subdomain overlaps, one can see the communication structure. The process that is responsible for the subdomain on the component grid has to send a share of its data to each of the processes that store a part of this subdomain in the distributed sparse grid. One can further observe that the number of communication partners is lower for the component grid with the less anisotropic domain decomposition.

This behavior is better comprehensible when looking at the one-dimensional structure of the domain decomposition. Figure 3.23 shows the communication structure in one dimension for four different configurations. In each configuration, the distributed sparse grid is in the top row and the distributed component grid is in the bottom row. The configurations differ in the number of processes being used to decompose the component grid in this dimension. The arrows indicate to which processes storing the distributed sparse grid each process in the component grid has to send its data. The numbers indicate the number of communication partners for each process in the component grid. One can observe that the number of communication partners, in one dimension, decreases when

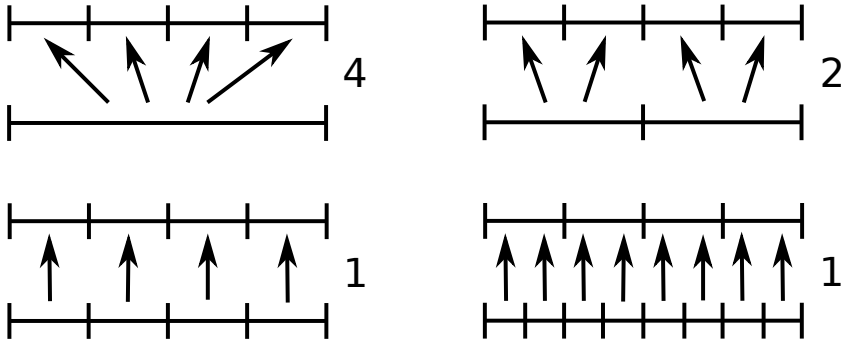

Figure 3.23.: One-dimensional communication structure. The separators indicate the domain decomposition. The arrows indicate messages. The numbers indicate the number of messages that are sent per process.

the number of processes for the component grid increases in this dimension. However, when the number of processes for the component grid is equal or higher than for the sparse grid, there is only one communication partner.

Assuming powers of two for the number of processes in each dimension, as it is used throughout this work, the number of communication partners for each process on the component grid in a particular dimension k can be formulated as

$$\left\lceil \frac{p_{\mathrm{dsg,k}}}{p_{\mathrm{cg,k}}} \right\rceil , \tag{3.11}$$

where $p_{\mathrm{dsg,k}}$ is the number of processes for the sparse grid in this dimension and $p_{\mathrm{cg,k}}$ the number of processes for the component grid, respectively. Then the total number of communication partners for each process in the component grid over all dimensions is

$$\prod_{k=1}^{d} \left\lceil \frac{p_{\mathrm{dsg,k}}}{p_{\mathrm{cg,k}}} \right\rceil . \tag{3.12}$$

For the case that is considered in this work where the sparse grid, as well as the component grid, is stored by the same number of processes (size of the process group), the maximum number of communication partners can be determined. For simplicity, the analysis neglects the fact that the sparse grid and the component grid are stored on the same set of processes and thus in some cases the sending and the receiving process are identical. The worst case can be created by choosing completely anisotropic domain decompositions for the sparse grid and for the component grid where the anisotropy is directed in different dimensions. Then each process has to send to each other process and one has the same all-to-all situation as in the case of variant 1, which was already identified to be unfavorable. However, if an isotropic domain decomposition is used for the sparse grid then the number of communication partners reduces significantly. Then the worst case, with respect to the number of communication partners, is that the domain decomposition of the component grid is completely anisotropic (all processes in one dimension, one process in all other

dimensions). In all except the anisotropic dimension it holds $\frac{p_{\text{dsg,k}}}{p_{\text{cg,k}}} = p_{\text{dsg,k}} = m^{\frac{1}{d}}$ and thus

$$\prod_{k=1}^{d} \left\lceil \frac{p_{\text{dsg,k}}}{p_{\text{cg,k}}} \right\rceil \leq \left(m^{\frac{1}{d}}\right)^{d-1} = m^{1-\frac{1}{d}}, \tag{3.13}$$

where m is the total number of processes. With a perfectly isotropic domain decomposition there are $m^{\frac{1}{d}}$ processes in each dimension. Unfortunately, this upper bound still increases with the number of processes, although less than linearly. A simple solution to this problem is to limit the anisotropy of the domain decomposition such that the ratio, per dimension, between the number of processes for the component grid and for the sparse grid does not exceed a certain threshold, i.e.,

$$\frac{p_{\text{dsg,k}}}{p_{\text{cg,k}}} \leq c. \tag{3.14}$$

Then the number of communication partners

$$\prod_{k=1}^{d} \left\lceil \frac{p_{\text{dsg,k}}}{p_{\text{cg,k}}} \right\rceil \leq C \tag{3.15}$$

does not increase with the number of processes. In practice, limiting the anisotropy of the domain decomposition is a fairly reasonable approach, because it is likely that a highly anisotropic domain decomposition is also undesirable for the performance of the application code.

A model for the runtime of the reduction step, similar to the model used to analyze variant 1, can be given. Assuming an equal distribution of the number of grid points N in the component grid (and also in the sparse grid) to m processes, each process stores N/m grid points. According to Equation (3.15) each process sends at most C messages of size $\frac{N}{Cm}$. Then the maximum communication time for the local reduction of a component grid can be estimated as

$$C(L + \frac{N}{BCm}) = CL + \frac{N}{Bm}, \tag{3.16}$$

with latency L and bandwidth B (unit: grid points per second). As the same set of processes is used to store the component grid and the sparse grid, each process also receives this number of messages. Assuming a round-based model, as in Section 2.7, where each process can either send or receive a message in each round, the communication time would double. In practice, one can assume that C is small compared to the number of processes and thus the communication time decreases linearly with increasing m.

This variant of the distributed sparse grid enables a scalable implementation of the local reduction step when a uniform domain decomposition is not possible due to restrictions of the application code. For application codes where both variants are possible one faces a trade-off. One can either use the best possible domain decomposition for each component grid in order to minimize the time for the computations of the component grids, at the price of a more expensive local reduction step. Or, one can use the uniform parallelization in order to minimize the time of the local reduction, at the price of potentially higher

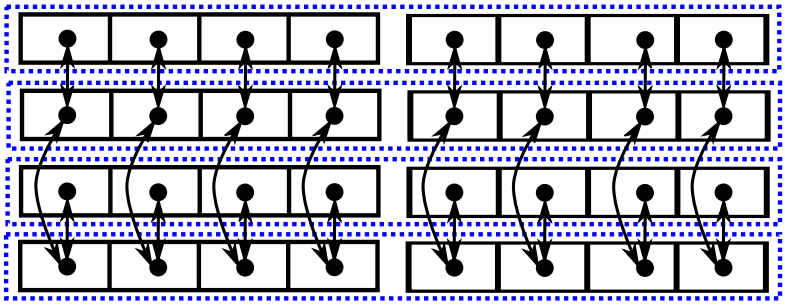

Figure 3.24.: Communication graph of the the global reduction step (assuming binomial trees for the *Allreduce* operation). The example shows four process groups. Each process group consists of two nodes with four processes each.

computation time for the component grids. The right choice depends heavily on the frequency of the combination step. If the component grids are combined only once in the end, or only very rarely, variant 3 might be more desirable. If the component grids are combined after each time step, then variant 2 might be the better choice. Variant 1 was included in this work for completeness and out of theoretical interest. In practice, it is, however, not useful for the local reduction.

3.3.4. Global reduction of the combined solution

Global reduction with process groups of equal size The global reduction step is identical for all variants of the distributed sparse grid. All processes which have the same position (the same local rank) in the process group store the same part (the same set of grid points) of the distributed sparse grid. These processes perform an `MPI_Allreduce` operation in order to reduce their local subdomain of the distributed sparse grid to the combined solution. This is visualized in Figure 3.24. This is done with a rather simple algorithm: The first step is to create a set of MPI communicators where each contains all processes which have the same relative position in their process group. Note that these communicators only have to be created once for the first combination step and can then be reused for subsequent combination steps.

Then, each process copies its local part of the distributed sparse grid into a buffer. This is done by iterating over the hierarchical subspaces in a prescribed order (e.g., lexicographic according to the index vector associated with each subspace). Then for each subspace, the hierarchical coefficients stored on this process are copied into the buffer (again in a prescribed order). If a particular hierarchical subspace does not exist in a process group (because no component grid was computed by this group which contains this subspace), the corresponding entries in the buffer are set to zero. This procedure makes sure that that the order of the hierarchical coefficients in the buffer is equal on all processes. Only then the buffer can be reduced efficiently with a single `MPI_Allreduce` operation. After the `MPI_Allreduce`, the hierarchical coefficients of the combined solution are available in the buffer of each process. In a last step, each process

copies the coefficients back from the buffer into its local part of the distributed sparse grid.

The cost of the global reduction step are analyzed in the following. With m processes per process group, the global reduction step consists of m `Allreduce` operations that are performed in parallel and individually of each other. The run time for the `Allreduce` operation can be estimated by (compare Section 2.7)

$$t_{ar} = 2\log(g)\left(L + \frac{N_{SG}/m}{B/M}\right) = 2\log(g)L + 2\log(g)\frac{N_{SG}/m}{B/M}, \tag{3.17}$$

with message latency L, number of process groups g and the size of the buffer N_{SG}/m (a sparse grid with N_{SG} grid points distributed onto m processes). When the network bandwidth of a compute node is shared by M processes, the effective bandwidth per process reduces to B/M. The time for `Allreduce`, and so the time for the global reduction, scales with the size of the process groups, but it increases logarithmically with the number of process groups.

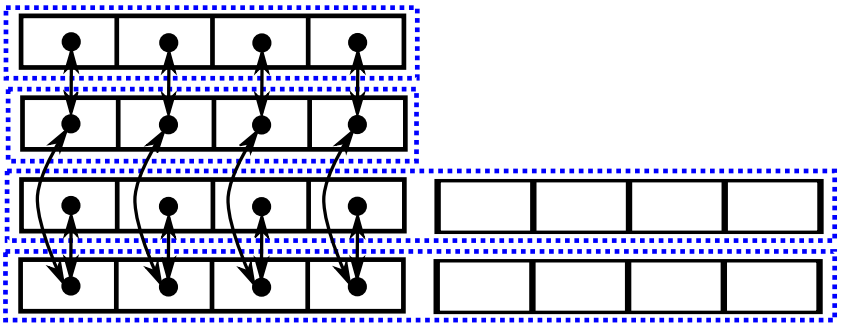

Figure 3.25.: Communication graph of the the global reduction step with process groups of different sizes. Only a subset of the processes is used to store the sparse grid and thus only this set contributes to the global reduction.

Global reduction with process groups of different sizes Although this was not considered for the experiments presented in this work, there might be situations when it is desirable to use process groups of different sizes. With increasing dimensionality also the difference in size between the smallest and the largest component grids in the combination scheme increases. A process group size that is appropriate for the largest component grids might be too large for smaller component grids and this might result in a bad parallel performance there. Especially for applications that do not scale very well, it could then make sense to use two or more different sizes of process groups in order to improve the performance of the computation step. Unfortunately, this increases the runtime of the combination step. So, again there is a trade-off between optimizing the performance of the computations or optimizing the combination step.

The above strategy for the global reduction with process groups of equal size can easily be adapted to process groups of different sizes. The idea is to store the distributed sparse

grid with a number of processes that is equal to the size of the smallest process group. Hence, for larger process groups only a subset of the processes in the process group is used to store the sparse grid. Only the processes that store a part of the sparse grid then contribute to the global reduction. This is visualized in Figure 3.25 with two different sizes of process groups. With variant 3 of the distributed sparse grid a local reduction to a smaller set of processes inside the process group can easily be realized.

Using less processes to store the distributed sparse grid increases the amount of data to be sent by each process during the `Allreduce` operation and thus increases the runtime. This can easily be seen in Equation (3.17). For the same number of process groups, reducing m by a certain factor increases the runtime by this factor. Furthermore, when only a subset of the processes in the process group is used to store the sparse grid, also the runtime for the local reduction increases. When reducing the number of processes used to store the sparse grid by a certain factor, the amount of data to be received by each of these processes increases by this factor.

3.3.5. Experimental results

The scalability of the distributed combination algorithm was investigated on up to $180,225$ (out of 185,088) cores of the supercomputer *Hazel Hen* (see Appendix A). At the time when the experiments were done, this system was the second fastest system in Europe and rank 8 on the Top500 list.

For the experiments the scale of the combination scheme was chosen so that the problem size is in a realistic range for future large-scale experiments with GENE. Setting $\vec{n} = (14, 6, 6, 8, 7)$ and $\vec{l}_{\min} = (9, 4, 4, 6, 4)$ results in 182 component grids. Note that the 76 largest component grids had between 74 GB and 83 GB (complex-valued numbers). A distributed sparse grid of variant 2 was used and equal parallelization for each component grid. Process groups of size `nprocs` = 1024, 2048, 4096 and 8192 were used. The number of process groups `ngroups` ranges between 1 and 128 (only powers of two were used). Furthermore, 22, 44, 88, and 176 process groups were used for the largest experiments with $180,225$ processes (one process for the manager). It was not possible to perform the experiments with less than 4096 cores, because the necessary amount of memory to store the component grids exceeded the main memory of the compute nodes.

Figure 3.26 shows the run times for individual substeps of the combination step: hierarchization, local reduction and global reduction. Furthermore, it shows the sum of the run times of hierarchization, local reduction and global reduction. Results for dehierarchization and scatter are not presented here, because they behave similar to their inverse counterparts, reduction and hierarchization and thus this would not provide further insight. The run times presented here are the average times per process group. The times presented for hierarchization and local reduction is the accumulated time to hierarchize and reduce all component grids assigned to the process group. The abscissa of the plots presents the total number of processes, i.e., `ngroups` $\times$ `nprocs`. The manager process is neglected here, because it does not participate in the combination step.

The run time of the hierarchization and the local reduction step scales perfectly with the total number of processes. The reason is that the number of component grids on a

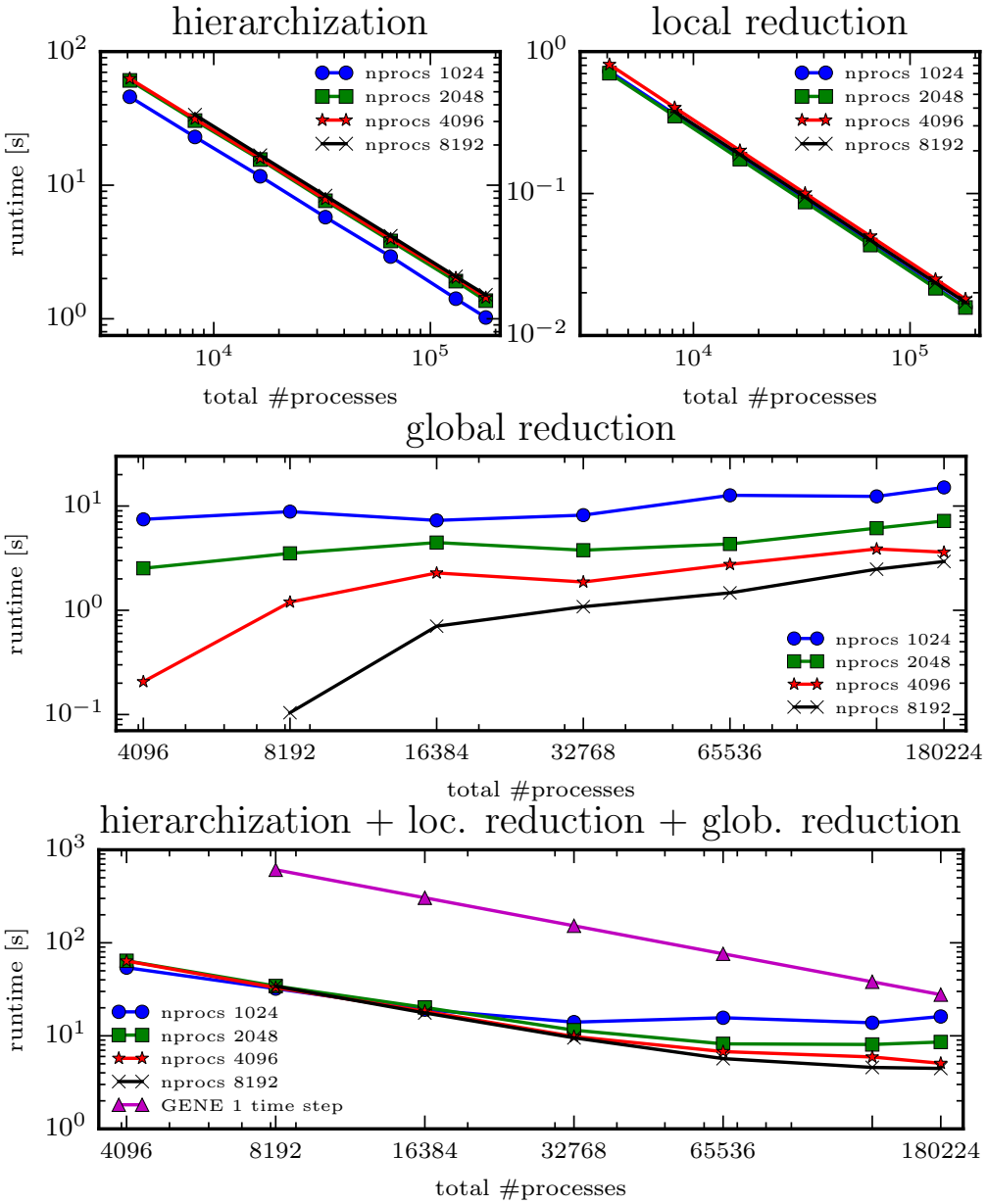

Figure 3.26.: Run times for the substeps of the distributed combination step for different sizes of the process groups. The bottom plot shows the sum of the times for hierarchization, local reduction and global reduction. A rough estimate of the computation time for one time step of GENE was included as well.

group decreases linearly with the number of process groups. In the same way the amount of grid points to be treated by each process decreases. However, when larger process groups are used the run time of the hierarchization step slightly increases. Unlike for

the local reduction, which really is a purely local operation, the hierarchization requires communication between the processes of the process group. The communication overhead is the reason for the observed effect. It does not scale perfectly with the size of the process groups. For details on the distributed hierarchization algorithm refer to Section 3.3.1.

The run time for the global reduction increases with the number of process groups, as predicted by the model. However, no strictly logarithmic increase of run time with the number of process groups can be observed. For example, with a process group size of 1024 the run time increased from 4 process groups (4096 processes) to 176 process groups (180,224 processes) by a factor of 2.2, whereas $\log(176)/\log(4) \approx 3.7$. Thus, the increase in runtime is actually better than logarithmic. This could be explained by the fact that other communication patterns than binomial trees might be used in actual implementations of `MPI_Allreduce` (see Section 2.7). For the cases `nprocs 4096` and `nprocs 8192` there was a strong increase of the run time when going from one to two process groups. This comes from the fact that with one process group no actual communication happens and so this was only the time to copy the grid points to the buffer and back. Furthermore, also as predicted by the model, the run time decreases for larger process groups, because the number of grid points decreases and so does the message size for each individual `MPI_Allreduce` operation.

The aggregate time for hierarchization, local reduction and global reduction scales until it is dominated by the time for the global reduction. For low numbers of process groups, the time for the hierarchization dominates. The time for the local reduction is so small in comparison to the time for the hierarchization that it can be neglected. The results demonstrate that strong scalability for the combination step can be achieved by adding more process groups until a certain point. After this point the time for the global reduction dominates, which does not scale with the number of process groups. This point, and thus the strong scalability limit of the combination step, can be shifted to higher process counts by using larger process groups. Thus, the experiments indicate that for the performance of the combination step, it is more desirable to use large process groups. However, increasing the size of the process groups is only sensible in the range where the application code scales.

A rough estimate for the computation of a single time step in GENE was included in Figure 3.26. This estimate corresponds to the average time per process group to compute all 182 component grids. However, not the individual times for all component grids were measured, as this would have wasted too many computational resources. Instead, one component grid of each of the five diagonals in the combination scheme was measured and the time was estimated under the assumption that all component grids on the same diagonal have the same run time. Of course, this is not very accurate, because the run-time does not only depend on the size of the component grids, but it is also significantly influenced by the shape of the grid (see Section 3.2). However, these number are accurate enough to present a rough guide how the run time of the actual computations relates to the time of the combination step. The measurements with GENE were done on 8192 processes with two species (the simulation scenario did not allow for a computation with only one species). Thus, one can use this as an estimate for the computation of one species on process groups with 4096 processes. The parameter file corresponding to the

simulation scenario used for the predictions can be found in Appendix B.2. Note that it was not possible to use less processes for the largest component grids, because of main memory limitations (GENE needs a lot more main memory than what is required to just store the grid). However, to provide a meaningful investigation of the performance of the combination algorithm also smaller process groups were used for the experiments. The experiments indicate that it is easily possible to combine after each time-step without adding significant overhead to the computations.

Directly computing this simulation scenario on the full grid with discretization level $\vec{n}$ with GENE would result in a computational grid with a resolution of $16384 \times 64 \times 64 \times 256 \times 128$ grid points (per species). This grid would have an almost 128 times higher number of grid points than it was used in the largest simulation run with GENE published so far [83], which had a resolution of $512 \times 512 \times 96 \times 48 \times 15$. The experimental setup corresponds to the combination of one species. However, in actual GENE simulations up to four species are used. Then the grid sizes, the computation time in GENE, and the time for the combination multiplies with the number of species.

3.4. Evaluation of the combined solution for post-processing

Algorithm 5: Evaluation of the combined solution

input :
$\vec{l}_{\text{eval}}$: discretization level of full grid on which solution is evaluated
$\vec{n}$: target level of combination scheme
c_u: combination coefficient which corresponds to component grid u
groups: set of process groups
grids(g): set of component grids on group g
g_k: process group to write the file

```
1  for g ∈ groups do in parallel
2      dsg_g ← dsg(n⃗) ;                                   // create distributed sparse grid
3      for u ∈ grids(g) do
4          u ← hierarchize (u) ;                          // hierarchize component grid
5          dsg_g ← dsg_g + c_u u ;                        // add hierarchical coefficients to sparse grid
6  dsg_g ← Σ_{g∈groups} dsg_g ;                           // global reduction of sparse grid
7  if g = g_k then
8      u_{l⃗_eval} ← createFullGrid(l⃗_eval) ;              // create full grid
9      u_{l⃗_eval} ← dsg_g ;                               // extract hierarchical coefficients from sparse grid
10     u_{l⃗_eval} ← dehierarchize (u_{l⃗_eval}) ;          // dehierarchize full grid
11     saveToFile (u_{l⃗_eval})
```

In order to plot or post-process the combined solution, it is usually more convenient to evaluate it on a regular grid than on a sparse grid. This is done in a dedicated evaluation step. In this step, the combined solution is evaluated on a full grid of a given discretization level $\vec{l}_{\text{eval}}$ and stored to a file. Usually, this would be used at the end of the simulation to evaluate the final solution with a desired resolution. Especially with time-dependent problems, another use-case might be to evaluate the solution at distinct

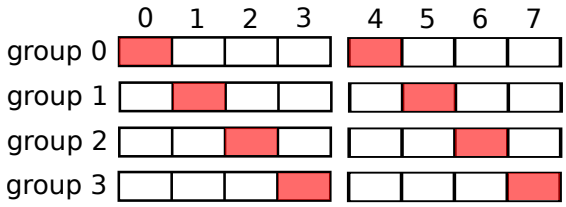

Figure 3.27.: Writing the combined solution to file using multiple process groups. The figure shows four groups which consist of eight processes and two nodes, each. The processes marked red write the output file. This way the number of nodes contributing to this operation can be increased by a factor of four compared to the case that only one group writes the output file.

points in time, e.g., after each combination step, to trace the development of the solution over time.

Algorithm 5 presents the essential steps of the evaluation procedure. Although the first part of the algorithm is (almost) identical to the combination step, there is a small but important difference. Here, the discretization level of the sparse grid really needs to be the target level $\vec{n}$ of the combination scheme. Unlike for the combination step where a coarser sparse grid can be used in order to save data transfers, this is not possible here. If the solution would be evaluated on the coarser sparse grid, one would loose the information of the largest hierarchical subspaces and thus obtain an approximation of lower quality.

After the global reduction only one process group is active. This group creates a distributed full grid with the desired resolution $\vec{l}_{\text{eval}}$. Then the hierarchical coefficients of the subspaces that are common on both, the full grid and the sparse grid, are copied into the distributed full grid. It is very likely that the full grid contains hierarchical subspaces that do not exist in the sparse grid. These coefficients are simply set to zero. Afterwards, the full grid is transferred into the nodal basis representation by dehierarchization. For these steps, the same algorithms as in the combination step are used.

Finally, the full grid containing the combined solution is saved to file in parallel using MPI-IO. This is a portable interface which is likely to provide very good performance when writing files in parallel on HPC systems that use a parallel file system, e.g. Lustre. Note that any attempt which tries to collect the data on one process and write the file in serial, e.g. with POSIX `write`, does not scale and is infeasible for the problem sizes targeted in this work.

Unlike for the combination step, where the combined solution has to exist on each process group after the global reduction step, for the evaluation it only needs to exist on one process group. Thus, the run time for the global reduction can be improved by reducing the data only to a single process group, i.e. using an `MPI_Reduce` instead of an `MPI_Allreduce`.

As it is more likely that the bottleneck is writing the file, rather than the global reduction, another potential optimization is to use more than one process group to write the file. This can be useful if more nodes than those available in one process group are

necessary to saturate the write rate of the parallel file system. The basic idea is to use the same number of processes to write the file, but distributed over more nodes. As the network bandwidth of a node is shared by all process, then the effective network bandwidth per process increases. In this case, of course the original `MPI_Allreduce` must be used for the global reduction and the steps after the global reduction in Algorithm 5 are executed on all process groups. As the full grid is then available on each process group, it is easily possible to only use every k-th process in each group, where k is the number of process groups being used to write the file. More specifically, of each group i of the k groups only the processes with local rank r would be active where it holds $(r + k)\%k = i$. This is visualized in Figure 3.27. Each of the marked processes writes the data corresponding to its local subdomain into the file.

3.5. Application interface

```
class Task {

public:
  inline const LevelVector& getLevelVector() const;

  virtual void init(CommunicatorType lcomm) = 0;

  virtual void run(CommunicatorType& lcomm) = 0;

  virtual
  DistributedFullGrid<DataType>& getDistributedFullGrid() = 0;
};
```

Figure 3.28.: The essential functionality of the task interface (the actual implementation contains more methods). Only the methods `init`, `run` and `getDistributedFullGrid` must be provided by the user in a derived class.

In the software framework the interface to the application code is realized with a `Task` class, which can be conveniently adapted to the specific requirements of the application. The responsibility of this class is to provide access to the simulation data while hiding the application's implementation details from the framework. The essential functionality of this class can be narrowed down to the few methods presented in Figure 3.28. The central element which defines a task is the level vector $\vec{l}$ which corresponds to the component grid it represents. Furthermore, the class offers methods to initialize the application and start the computations and a method which provides access to the actual data structure which stores the distributed component grid. Although the actual implementation contains many more methods than listed here, most of them are independent of the application code. Only a few absolutely necessary functions have to be provided by the user in a derived class.

The key part for the user is to provide efficient access to the simulation data. In the best case, the application code can directly work with the data structure for the distributed component grid provided by the software framework. Then no additional transformations of the data, which might involve additional communication between the processes, are necessary. However, this is very unlikely for most application codes that already exist, because they use their own data structures. If this is the case, the user has to provide the logic to convert the simulation data to the format used by the data structure for the distributed component. For the sake of efficiency it is highly desirable that these conversions are done directly in main memory at runtime. If this is not possible, as it might be the case for some proprietary codes, the alternative would be to realize this by reading and writing input and output files. Although this is likely to be very inefficient and not feasible for very large problem sizes, this option is mentioned here to emphasize that the users of the software framework have maximal flexibility on how they adapt their application code to the software framework. These conversions occur before and after each combination step and before the evaluation. Thus, how efficiently the data is accessed becomes more and more important, the more often these steps occur. If one wants to combine after each time step it is crucial to provide very fast access to the simulation data. However, if one only needs to combine once in the end, the overhead to read in the data from files might not be an issue.

3.6. Related work

A (fault-tolerant) parallel combination technique was first published by our international collaboration partners M. Ali, B. Harding, P. Strazdins, M. Hegland, et al. [7, 80, 6, 8, 81, 79]. Their work, as well, targets applications that solve time-dependent PDEs and employ distributed memory parallelization.

Their parallelization concept uses a distinct process group for the computation of each component grid. Load balancing is then achieved by using different sizes of process groups, i.e., for each diagonal in the combination scheme a different size is used for the process groups that compute the grids of this diagonal. As already discussed in Section 3.2, even component grids that are on the same diagonal can have significantly different runtimes and thus it is difficult to achieve good load balance with such an approach. Their load balancing measurements with the application GENE show considerable load imbalances between the different component grids on the largest diagonal in the combination scheme (compare Figure 6.3 in [6]). This further supports this claim.

The main difference of their approach is the implementation of the combination step. It consists of a reduction and a scatter step, as it is used in this work. However, the reduction step (and the scatter step) is not further divided into a local and a global substep, but instead performed only in one global step. The concept of this reduction step is similar to variant 3 of the local reduction (see Section 3.3.3). However, the sparse grid that stores the combined solution is distributed over almost all available processes. In such a configuration, the number of messages that each of these processes receives increases linearly with the number of component grids. Furthermore, the size of the

messages decreases linearly with the number of processes used to store the sparse grid. Thus, with increasing number of component grids and increasing number of processes, this results in a high number of very small messages, which might inhibit the scalability of the combination step.

Their implementation, however, was focused on fault-tolerance mainly, rather than on achieving maximum scalability. Their experiments were performed with relatively small problem sizes and a low number of component grids (high minimum levels were used). Results were only presented with around 3000 processes and only in two and three dimensions. Thus, the experiments are hardly comparable to the experiments in this work which target higher dimensionalities, significantly larger problem sizes and extreme parallelism (up to $180,225$ processes). The differences of the two approaches concerning the fault-tolerance mechanisms are discussed in Section 4.

4. A scalable fault-tolerant combination technique

One of the major issues of future exascale systems is that there is a lack of appropriate resilience solutions to deal with the increasing rate at which faults will occur [19]. The reasons for this relate to both hardware and software. On the hardware side, the increase in performance is expected to be achieved by an increasing number of components. Assuming the failure rate of a single component to be the same as those of current systems, having more components will already result in a higher failure rate for future systems. However, with an increasing integration level of microprocessors (more and smaller transistors) also the single components might have higher failure rates. The market for dedicated HPC hardware is very small and thus HPC systems are commonly built of commodity hardware that do not require high reliability. Thus, it is questionable whether the hardware vendors will compensate the increasing fault rates with additional (costly) logic for the detection and correction of faults [19]. On the software side, as the compute platforms become more complex and heterogeneous, also the whole software stack, from the system level up to the scientific applications, will become more complex and thus more prone to errors.

The increasing probability of failure with an increasing number of processes is a new limiting factor for the scalability of parallel applications. Even if the overhead for the fault recovery scales perfectly with the problem size (which is already an optimistic assumption), an increasing number of faults will result in higher total overhead for the fault recovery. Thus, the design of scalable and efficient fault tolerance mechanisms is crucial to ensure the scalability of parallel applications on future HPC systems.

So far, the standard approach to handle (process) faults in parallel applications are *checkpoint/restart* strategies [19]. Here, the current state of the computations is periodically saved to a more reliable medium than the hardware used for the computations, typically the central file system of the HPC systems. When the program crashes due to faults, the computations are restarted from the last checkpoint. The main drawback of such an approach is that creating and loading the checkpoint is potentially time- and resource-intensive and the corresponding overhead grows with the problem size. Thus, refining these techniques in order to make them applicable to exascale systems is a topic of ongoing research. Several methods are being explored to reduce the size of checkpoints [85, 60, 70] or to reduce the checkpoint time, e.g, by in-memory checkpointing [87], or multi-level checkpointing [72].

Another approach to deal with process faults is *replication*. In the most basic version of replication the program is simply started multiple times, preferably even on separate systems. As this comes with a very high resource overhead, such extreme approaches

usually are only used for systems with very high failure rates or if execution in real time is crucial (e.g., simulations for weather forecast). Ongoing research focuses on reducing this overhead using more fine-grained replication strategies that do not replicate the whole program [25].

An interesting alternative to these system-level approaches is *algorithm-based fault tolerance*, where the properties of the numerical algorithms are exploited in order to detect and recover from faults. In the context of exascale systems this research area experienced considerable progress within recent years. Examples from the area of iterative methods can be found in [77, 20]

The combination technique is a very promising candidate for algorithm-based fault tolerance due to the inherent redundancy (the same problem is solved with different discretizations). The concept for a fault-tolerant combination technique (FTCT) was proposed in the EXAHD project proposal and was first published in [45] by our international project partners. This was the basis for the massively parallel implementation of the fault-tolerant combination presented in this work. So far, this is the only implementation of the FTCT that has been applied to a five-dimensional problem and demonstrated to scale up to 65536 cores of a supercomputer. Furthermore, in order to be able to perform experiments with the FTCT on a supercomputer, a fault simulation software layer was developed. The here presented implementation of the FTCT is the result of close collaboration with Alfredo Parra Hinojosa within the project EXAHD. The investigations presented in the following were published in [48].

A parallel version of the fault-tolerant combination technique was first published by our collaboration partners M. Ali, B. Harding, P. Strazdins, M. Hegland, et al. (see Section 3.6). It is based on the same idea for the fault-tolerant combination technique [45]. In [48], we have extended the approach to higher dimensions and have demonstrated the scalability to significantly larger problem sizes.

In the context of this work the term *fault* means a *fail-stop*, i.e., a process fails (crashes) and all state in the process is lost [19]. The FTCT algorithm relies on system level methods to detect such a fault. For the applications addressed in this work this functionality will be provided by a fault-tolerant MPI implementation in the future. The other category of faults – *fail-continue*, where a failure occurs, but the process continues – is not so easily detectable. An example for such so-called *silent faults* are undetected and uncorrectable memory errors (so-called *silent data corruption*) that can potentially destroy the whole solution of a numerical simulation. As silent faults are expected to become the most important type of faults for future exascale systems [19], there has been considerable research progress in this area within the recent years.

Fortunately, the inherent redundancy of the combination technique cannot only be exploited to correct faults, but also to detect faults caused by silent data corruption. First investigations of this topic are presented in [54]. Once a silent fault has been detected, the combined solution can be recovered with the same mechanisms that are presented in the following.

4.1. Fault-tolerant combination technique

Consider a combination technique where each component grid is computed by a different set of processes or compute nodes. Now assume that a number of these processes or nodes crashed while computing the component grids, e.g., due to hardware faults. Hence, some of the component grids are lost. The trivial approach to deal with this situation would be to simply recompute those missing component grids, either on a new set of processes/nodes (under the assumption the systems allows us to request new computational resources at runtime) or by using the remaining healthy processes/nodes. This very simple form of a fault-tolerant combination technique has already an advantage over the computation of a single large full grid solution: if one loses only a small subset of the computational resources, e.g., a single process/node, one only has to recompute a small subset of the component grids. In the case of the full grid, one would have to recompute everything.

For time-dependent problems (compare Section 2.6), which are the focus of this work, one does not even have to recompute the missing component grids from the very beginning, i.e., from $t = t_0$. Instead, one would initialize the missing component grids with the combined solution from the last successful combination step. Thus, one only has to recompute the lost component grids for the current outer time step ΔT. This approach essentially is a form of the *checkpoint/restart* mechanism, where the combined solution of the last successfully computed outer time step serves as a kind of in-memory checkpoint. As the creation of this "checkpoint" happens in the course of the combination step, no further overhead is necessary if one accepts a small loss in accuracy. The reason is that efficient implementations of the combination step use a sparse grid of reduced resolution (compare Sections 2.7 and 3.3) in order to reduce the communicated data volume. Thus, if one desires to produce a "checkpoint" with the full resolution of the sparse grid, one has to accept an overhead for the combination step.

In the context of a massively parallel combination technique recomputing all the missing components grids has an adverse effect on the load balance. Consider a case where, for example, only one component grid out of several hundred has to be recomputed. All the processes assigned to the other, healthy, component grids have to wait until the recomputation of this single component grid is finished. Note that it is also more likely that a large component grid fails than a small one and thus the time for the recomputation might be significant. Independently of the actual distribution of computational resources to the component grids, either a larger number of processes is used for the large component grids than for the smaller ones, or the time to compute the larger ones is longer. In both cases the probability that one of the processes crashes while computing a large component grid is higher than for a small one. Furthermore, in the combination scheme there are more large grids than small ones (compare Figure 3.4). Repeatedly having faults in a significant number of outer iterations will add substantial overhead in compute time being spent to recompute the lost component grids.

The fault-tolerant combination technique technique [45] (see also [44, 42, 46, 67, 47, 43]) removes the overhead for recomputation by a lossy approach. The idea of this approach is as follows: When a subset $\mathcal{J}$ of the component grids in the index set $\mathcal{I}$ has been lost

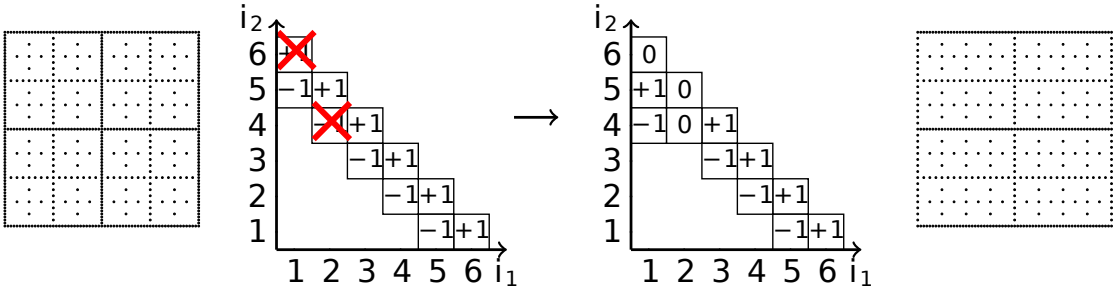

Figure 4.1.: A fault-tolerant combination technique with two faults and the alternative set used to correct the combined solution.

due to faults, one tries to find another combination scheme using only the remaining healthy grids in $\mathcal{I} \setminus \mathcal{J}$ in order to correct the combined solution. This is visualized in Figure 4.1. Here, the two crossed out component grids are missing and a new combination scheme avoiding these two grids was chosen. Note that in the new scheme a grid of a lower diagonal is used (it has now a non-zero coefficient) which had a zero coefficient in the original combination scheme. Further note that the underlying sparse grid of the combination scheme has fewer grid points and so one would also expect the accuracy of the combined solution to decrease. It was found, however, that the impact on the accuracy is only very small. Together with a very low overhead for the correction of the combined solution, this variant of the combination technique is a very promising approach to enable fault-tolerance for massively parallel applications on future HPC systems.

The essential question is now how to find such a new combination scheme which excludes the lost component grids and at the same time preserves the best possible accuracy. This is described in detail in [42] and will be summarized in the following. A first condition such a combination scheme has to fulfill is that it must be valid. A combination scheme, given as a set of level vectors $\mathcal{I}$ and corresponding combination coefficients (compare Section 2.5)

$$\sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} u_{\vec{l}}(\vec{x}) \,,$$

is valid if for all $\vec{l} \in \mathcal{I}$ it holds

$$\left(\sum_{j \in \mathcal{I}, \vec{j} \geq \vec{l}} c_{\vec{j}} \right) \in \{0, 1\}. \tag{4.1}$$

Essentially this condition makes sure that combination coefficients are appropriately chosen such that the so-called inclusion-exclusion principle is fulfilled. This means, if two functions $u_{\vec{i}} \in V_{\vec{i}}$ and $u_{\vec{j}} \in V_{\vec{j}}$ are added, their intersection space $V_{\vec{i}} \cap V_{\vec{j}}$ would contribute twice to the result and thus has to be subtracted, $u_{\vec{i}} + u_{\vec{j}} - u_{\vec{i} \wedge \vec{j}}$ The index of the intersection space is defined as $\vec{i} \wedge \vec{j} := (\min(i_1, j_1), \ldots, \min(i_d, j_d))$. This condition holds

for the different variants of the classical combination scheme introduced in Section 2.5, but it clearly does not hold any more when some of the component grids are missing due to faults (and thus removed from $\mathcal{I}$ without adapting the combination coefficients of the other grids).

New coefficients for the reduced set $\tilde{\mathcal{I}} := \mathcal{I} \setminus \mathcal{J}$ (with $\mathcal{J}$ being the set of lost grids) have to be found such that the corresponding combined solution provides the best possible approximation. Except for the case where the missing component grids are only on the highest diagonal, the coefficients cannot be determined by Equation (2.26), because the admissibility condition (2.25) is not fulfilled any more. For the general case, in [42] it was proposed to find the new coefficients by solving an optimization problem, the *generalized coefficient problem* (GCP). The underlying assumptions that led to this optimization problem are related to the error bounds assumed for sparse grid interpolation. Although these assumptions might not be fulfilled in general, this is the best choice without having any further knowledge on how the individual component solutions contribute to the overall accuracy of the combined solution.

The main challenge of solving the GCP is that this problem is NP-hard and in some cases, finding $\tilde{\mathcal{I}}$ can take very long times, especially in high dimensions. A workaround of this issue is to avoid some choices of $\tilde{I}$ that are known to be unfavorable for the solution of the GCP, by recomputing some of the lost component grids. Furthermore, the GCP can result in non-zero combination coefficients for grids having zero coefficients in $\mathcal{I}$. Usually, these grids are not considered for computation as they do not actually contribute to the combined solutions. In order to deal with these issues, in [42] the following strategy was proposed:

- If component grids on the two highest diagonals of the combination scheme, $\mathcal{D}_{\mathcal{I},\max}$ and $\mathcal{D}_{\mathcal{I},\max-1}$, are affected by faults, solve the GCP to find a new combination scheme.

- If component grids on the other main diagonals of the combination scheme, i.e., on the diagonals $\mathcal{D}_{\mathcal{I},\max-d} < \mathcal{D}_{\mathcal{I},a} < \mathcal{D}_{\mathcal{I},\max-1}$, are affected by faults, these grids are recomputed. This avoids cases where the solution of the GCP is very costly.

- Additionally, to the grids on the main diagonals of the combination scheme also the two diagonals $\mathcal{D}_{\mathcal{I},\max-d}$ and $\mathcal{D}_{\mathcal{I},\max-d+1}$ are computed. These extra grids have a zero combination coefficient in the regular combination scheme. However, they might get a non-zero coefficient in the alternative combination scheme and thus must be readily available.

The idea behind this strategy is to accept a certain overhead for the recomputation of the grids below the two highest diagonals in order to simplify the solution of the GCP, which otherwise could potentially take much longer. Furthermore, one accepts additional overhead for the computation of the extra grids below the main diagonals (those which have zero coefficients in $\mathcal{I}$), which do not contribute to the combined solution when no fault occurs. However, the lower the diagonal, the lesser is the number of component grids on this diagonal and the smaller is the size of these grids. Thus, the overhead to

compute the extra grids is relatively low in comparison to the time it takes to solve the grids on the main diagonals. Applying the fault-tolerant combination technique to GENE, in [55] it was confirmed that this is a reasonable strategy. Here, for a large-scale problem with GENE, this overhead was estimated to be only 3–4%. Furthermore, it was found that the overhead for the recomputation of the lost component grids that are positioned below the two highest diagonals is very small in comparison to the time it would take to recompute all component grids.

4.2. Implementation

Algorithm 6: Fault-tolerant parallel combination technique.

```
input :
        t_0, t_end: start and end time
        ΔT: outer time step size
        I: set of component grids
        c_u: combination coefficient which corresponds to component grid u
        groups: set of process groups
        grids(g): set of component grids assigned to group g
1  distribute component grids in I to process groups and set initial conditions
2  t_i ← t_0
3  while t_i ≤ t_end do
4      for g ∈ groups do in parallel
5          for u ∈ grids(g) do
6              u ← solve (u, ΔT) ;                          // solve u for next outer time step
7      if faults detected then
8          recover () ;                                     // recover from faults using the FTCT
9      for g ∈ groups do in parallel
10         dsg_g ← dsg(n⃗) ;                                  // create distributed sparse grid
11         for u ∈ grids(g) do
12             u ← hierarchize (u) ;                        // hierarchize component grid
13             dsg_g ← dsg_g + c_u · u ;                    // add hierarchical coefficients to sparse grid
14     dsg_g ← Σ_{g∈groups} dsg_g ;                         // global reduction of sparse grid
15     for g ∈ groups do in parallel
16         for u ∈ grids(g) do
17             u ← dsg_g ;                                  // extract hierarchical coefficients from sparse grid
18             u ← dehierarchize (u) ;                      // dehierarchize component grid
19     t_i ← t_i + ΔT
```

The software framework for the massively parallel combination technique (see Section 3) was extended to include the fault-tolerance mechanisms described in the previous section. Algorithm 6 shows where the detection and recovery of faults is embedded in the parallel combination technique algorithm. In this version of the implementation only faults that occur during the computation of the component grids are detected, because it is assumed that this is the most time consuming part and thus here is the highest probability that a fault occurs. However, one can easily repeat the fault detection and recovery procedure after the combination step in order to also protect the algorithm from faults that occur

during the combination step. To protect the algorithm from the unlikely event that of all processes exactly the manager process fails, one could use a replication strategy, e.g., a second manager process on a different compute node.

The algorithms for the detection of and recovery from faults are based on the ULFM [13] specification, which is currently the most mature implementation of a fault-tolerant MPI. It extends the MPI standard by the necessary functionality to detect crashed processes and to exclude these processes from any future communication.

In the parallelization concept described in Section 3 there are two ways to detect a failed process. Either the master process in the process group detects that a process in its own group has failed and reports this to the manager, or the manager detects that the master process of a particular group has failed. The former case is implemented by a call to the function `MPI_Barrier` by all processes of the group. If a process in the group has failed, this function will return an error code on all living processes. The master process can then send an according signal to the manager process. In the latter case, the manager waits for a message (`READY` signal, see Section 3.1) from the master process that signals that the process group changed its current state. This is implemented with a non-blocking `MPI_Irecv` and an according `MPI_Wait`, which will return an error code if the master process has failed.

If the application code is not prepared for errors in MPI calls (presumably this is the case for many codes) and has no predefined exit strategy, after a process failure it might happen that some of the alive processes of the process group block forever and never reach the barrier which is used to detect a fault. If one observes such a behavior with a specific application code one can fall back to a failure detection based on timeouts. If the manager does not receive the `READY` signal of a particular process group within reasonable time (relative to the other groups), it simply assumes that this group has failed.

After a fault has been detected in a process group, the whole process group is considered to have failed and is removed. This means that all component grids computed on this group are lost. The recovery procedure is then performed as follows:

1. All the MPI communicators that contain processes of the failed process group are newly created in order to exclude the processes of the failed group from future communication. This is implemented with the ULFM functions `MPI_Comm_revoke` and `MPI_Comm_shrink`.

2. Find a new combination scheme which excludes the lost component grids as described in Section 4.1.

3. Distribute the lost component grids to the living process groups. If the application code requires to run some initialization routines before the actual computations, e.g., set up data structures, etc., this is triggered now.

4. Sometimes it is necessary to recompute some of the lost component grids. If this is the case, initialize these component grids with the last combined solution (which is still available from the last successful combination step) and compute them for the required time interval.

The corrected combined solution will be computed during the next combination step (lines 9 to 18 in Algorithm 6). The only thing that is different in the algorithm for the combination step following the recovery procedure, compared to the "regular" combination step, is that the combination coefficients obtained in step 2 of the recovery procedure are used. Thus the lost component grids get a zero coefficient and do not contribute to the combined solution. Note, however, that after the combination step these grids have been set with the (corrected) combined solution as well (line 17 in Algorithm 6). In the following outer iterations the full original combination scheme will be computed (unless more faults occur, of course).

The strategy of always removing the whole process group whenever a fault occurs obviously is a waste of computational resources. However, this first implementation of a fault-tolerant and massively parallel combination technique served the sole purpose of demonstrating the feasibility of such an approach on an HPC system. The question of how this strategy can be refined strongly depends on the actual design of fault-tolerant MPI implementations on future HPC systems. Is it possible to request new computational resources at runtime after a fault occurred? Then one would be able to restore the process group (the component grids would then still be lost, but one does not loose the process group as a computational resource). If not, after a second group has failed one could rearrange the remaining healthy processes of both groups into a new group. This way, as long as the total number of crashed processes/nodes does not exceed the size of one process group, the number of lost resources never exceeds the size of a single process group. For a large-scale problem where tens or hundreds of process groups are used, the fraction of resources wasted by removing a single process group is tolerable. Furthermore, if process groups of different sizes are used, another strategy would be to shrink a process group to a smaller size after some of their processes have crashed.

4.3. Fault simulation software layer

The implementation of the fault-tolerant parallel combination technique is based on ULFM and has been successfully tested on a local system. Unfortunately ULFM (or any other fault-tolerant MPI implementation) was not yet available on the HPC systems we had access to. Thus, in order to be able to perform experiments with the fault-tolerant combination technique on a large-scale system, a fault simulation software layer was implemented.

This software layer is placed between the system's native MPI implementation and the application code (see Figure 4.2). It offers the interface of ULFM to the application code and enables the user to simulate crashed processes that then can be handled by the fault-tolerance mechanisms of the application code. Note that in this context application code means just any code that wants to use the features of ULFM. The fault simulation layer is implemented in a general way and is an independent software library. It is not specifically tailored to the combination technique software framework or to the simulation codes, DUNE and GENE, used in this work. Note that this software layer is only a tool to run the experiments with simulated faults on a system where no fault-tolerant

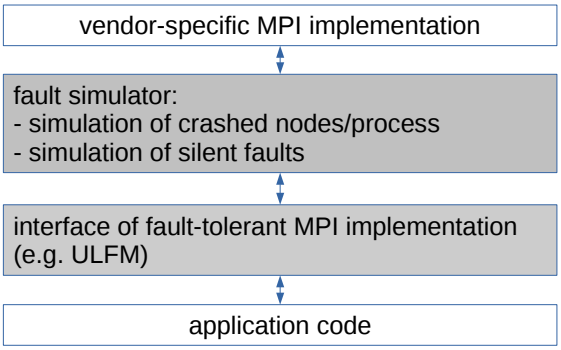

Figure 4.2.: Concept of the fault simulation software layer.

MPI implementation is available. It is not intended to, and it cannot, replace an MPI implementation which can handle actual process faults.

The fault simulation layer consists of two parts. The interface layer offers the essential functionality of ULFM. This means it emulates the external behavior by, e.g., returning appropriate error codes. The internal implementation of these functions, however, is different from ULFM, since the fault simulation layer only has to deal with simulated faults and not actual process faults. The functions that are supported in the fault simulation layer are the most common point-to-point and collective operations plus some of the new fault-tolerance functions, such as `MPI_Comm_shrink` and `MPI_Comm_revoke`.

The simulation layer offers the possibility to simulate crashed processes. The user can trigger a process to crash virtually by calling a `kill_me` function. The process then stops its normal operation and goes into a an idle state where it handles only background operations of the fault simulation layer. At this stage, this process can be detected as having failed by the other processes. For example, if this process would be the destination of an `MPI_Send`, or if it would participate in a collective operation, such as `MPI_Allreduce`, these functions would return on the other processes with the appropriate error message, e.g., `MPI_ERR_PROC_FAILED` according to the ULFM specification.

Another feature of the simulation layer is the simulation of silent faults by directly manipulating the MPI messages. This is done by injecting a number of bit-flips according to some probabilistic model into the data passed to, e.g., `MPI_Send`. The models used to manipulate the messages can be given by the user. The advantage of this approach is that the logic to produce silent faults was moved out of the application code and into the fault simulation layer. This way, it can be easily reused for other applications.

The basic idea behind the implementation of the fault simulation layer is visualized in Figure 4.3 (left) at the example of two processes that call a matching pair of `MPI_Send` and `MPI_Recv`. Before the actual message is sent, the sender sends an *alive message* to the receiver. Conversely, the receiving process sends an *alive message* to the sender. This way both processes know that the respective other process is alive and then start the normal transfer of the message. Figure 4.3 (right) visualizes the case that the receiving process is dead. Any dead process is not actually dead, but instead it remains in an idle background state. Whenever it receives an alive message it would answer this message

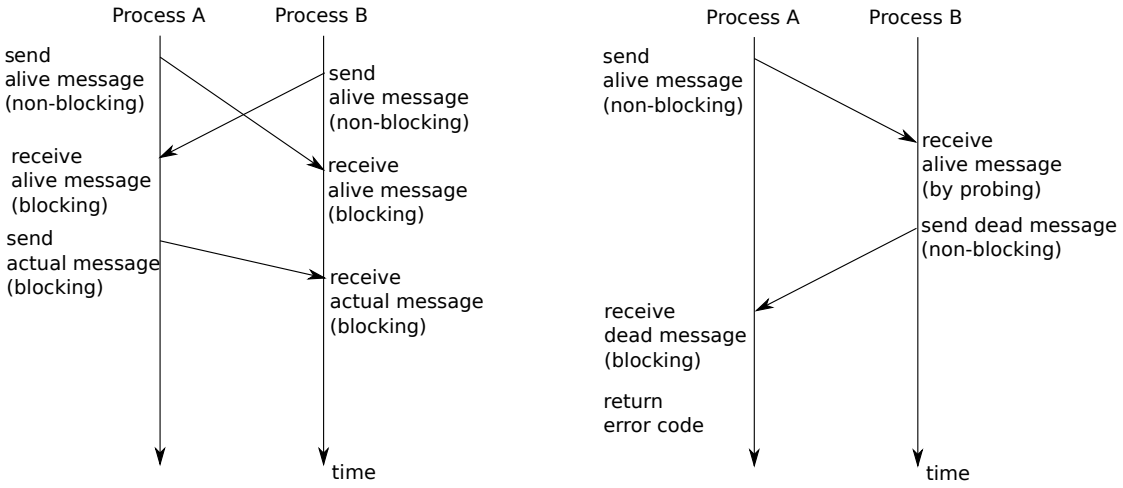

Figure 4.3.: **Left:** Blocking send to an alive process. **Right:** Blocking send to a dead process. Adapted from [84].

with a *dead message*. Now the sender knows that its counterpart is dead and returns an appropriate error message. A similar scheme is employed in the case of non-blocking operations and collective operations. However, as one has to deal with consistency issues here, the implementation is much more complex. For details on the implementation of the fault simulation layer refer to the corresponding Master's Thesis [84].

Obviously the alive messages add overhead to the communication operations. This overhead was observed to be significant for very small message sizes. Here the times to send the alive messages are in the same order of magnitude as the actual data transfers. However, it becomes negligible for large messages, where the time for the data transfer is the dominating factor. This was observed for both point-to-point and collective communication operations. Thus, the fault simulation layer has an adverse effect on the runtime and scaling behavior of the application. This problem was solved by offering selective fault tolerance in the interface of the fault simulation layer. This means there are two versions of an MPI function, e.g., `Sim_FT_MPI_Send` and `MPI_Send`. In the first case, the fault-tolerant version experiencing the overhead of the simulation layer is used. In the latter case, the MPI function is called directly. In the case of simulated faults, the user can control when faults can happen and when not. Thus, it is not necessary to use the fault-tolerant version of the MPI functions everywhere, but instead only in a limited number of places. For the experiments presented in this work the fault tolerant versions of the MPI calls were only used to detect the faults and to adapt the communicators, but not in the simulation code or other performance critical parts. This way, the impact of the fault simulation layer on the overall performance and scalability was barely noticeable. However, if one does not wish to choose manually whether the fault-tolerant version of an MPI function is used, or the regular one, there is a compiler switch available to pass all MPI calls through the fault simulation layer.

Even if ULFM would be available on the desired system to run the experiments, the

fault simulation layer still has an important advantage. Most supercomputer systems use a native MPI implementation that is specifically optimized to the system's architecture. These native implementations often perform significantly better than OpenMPI, which is the basis of the ULFM implementation. Thus, the fault simulation layer enables to test a fault-tolerant algorithm and still to obtain the performance of the native MPI implementation. One final advantage is the portability of the fault simulation layer. It runs out-of-the-box with any standard MPI 3.0 implementation and has no other dependencies. This way the experiments can be easily repeated on different systems without having to make sure that ULFM is available in each single case.

4.4. Experiments and results

The test problem used for the experiments was the d-dimensional advection-diffusion equation

$$\begin{aligned} \partial_t u - \Delta u + \vec{a} \cdot \nabla u &= f \qquad \text{in } \Omega \times [0,T) \\ u(\cdot, t) &= 0 \qquad \text{in } \partial\Omega \end{aligned} \tag{4.2}$$

with $\Omega = [0,1]^d$, $\vec{a} = (1,1,...,1)^T$ and $u(\cdot,0) = e^{-100\sum_{i=1}^{d}(x_i-0.5)^2}$. This problem was solved with the well established PDE-framework DUNE-pdelab [14] using the finite volume element (FVE) [11] method on rectangular d-dimensional grids for the spatial discretization. Unlike with the classical finite volume approach where the degrees of freedom (DOFs) are positioned at the cell corners, the FVE method uses node-centered control volumes. This way the DOFs in DUNE are positioned at the grid points of the component grids used for the combination technique. A simple explicit Euler scheme was used for the time integration. This problem was chosen, because its numerics are well understood and it allows to perform the experiments with arbitrary dimensionality.

The function *solve* in Algorithm 6 corresponds to calling DUNE in order to solve the corresponding component grid for a single outer time step ΔT. In each outer iteration only one time step was computed on the component grids and thus $\Delta T = \Delta t$ For these experiments it was not of interest to investigate the impact of the outer time step size on the accuracy of the combined solution and thus this choice was just convenient. DUNE was executed in parallel using all processes of a process group and a regular geometric domain decomposition. As DUNE uses its own data structures, the data has to be copied and converted from the component grid into the data structures of DUNE, and vice versa, before and after each call to DUNE. A process failure was simulated during the *solve* function via the `kill_me` function of the fault simulation software layer. Which process fails in which outer iteration was specified a priori in a parameters file.

The theoretical convergence of the fault-tolerant combination technique has been studied before [42, 55]. The following convergence study confirms these results in a five-dimensional setting (so far, it was only studied in up to three dimensions). The scalability study in Section 4.4.2 demonstrates that the implementation of the fault-tolerant combination technique presented in this work can handle process faults in a large-scale setup with only a small overhead. So far, this is the only implementation for

which experiments with the fault-tolerant combination technique have been reported with such a large number of processes.

4.4.1. Convergence

The convergence of the fault-tolerant combination technique was studied by increasing the target resolution $\vec{n}$ of the combination technique and for each case computing the error of the combined solution to a highly resolved reference solution that has been computed on a full grid. This was repeated with different numbers of lost component grids in order to study the effect on the accuracy of the combined solution.

The computations for each target level $\vec{n}$ were repeated with different numbers of process groups and in each case one random group failed. As for each target level there is the same number of component grids, in each case this results in different percentages of lost component grids. Thus, this procedure is representative for multiple numbers of failed groups (it does not matter if one out of two groups fails, or two out of four, the number of lost component grid is approximately equal). With the component grids being homogeneously distributed among the process groups according to their computational costs (see Section 3.2), which exact group failed had no noticeable impact on the final accuracy of the combined solution. Thus, the following results correspond to a single simulation run for each target level and number of process groups. In all of the experiments a process was set to fail during the second outer iteration (so that component grids have been distributed to all groups before). For the error calculation the combined solution was interpolated to the reference grid and the relative l_2-error was computed as

$$\frac{\|u_{\vec{n}}^{(c)} - u_{\text{ref}}\|_2}{\|u_{\text{ref}}\|_2} \tag{4.3}$$

The error was computed at the end of the simulation, $t = 0.1$ in 2D and $t = 0.05$ in 5D. A common time step size $\Delta t = \Delta T$ was used for each component grid, with values of 10^{-4} in 2D and 10^{-3} in 5D. Thus, the component grids were combined after each time step. Furthermore, after each time step a fault could be detected and corrected.

The convergence of the fault-tolerant combination technique in 2D is shown in Figure 4.4 (left). Here, $\vec{l}_{\text{min}}$ was fixed to $(3, 3)$ and the target level $\vec{n}$ ranged from $(6, 6)$ to $(10, 10)$. The full grid reference solution was computed with level $\vec{n} = (11, 11)$. The legend indicates the number of process groups that was used for the experiments and the resulting (approximate) percentage of lost component grid when one of the groups fails. The recovered combined solution was only slightly (1%–3%) worse than in the case without faults, even when half of the component grids fail. The difference is more visible for the experiments in five dimensions, shown in Figure 4.4 (right). Here, $\vec{l}_{\text{min}}$ was set to $(3, 3, 3, 3, 3)$ and $\vec{n}$ ranged from $(4, 4, 4, 4, 4)$ to $(6, 6, 6, 6, 6)$. The full grid reference solution was computed with level $\vec{n} = (6, 6, 6, 6, 6)$. With four process groups the accuracy of the recovered combined solution was between 3% and 9% worse than the case without faults. With two groups this number ranged between 9% and 17%. Although a 17% higher error might not be an acceptable outcome for an actual simulation (and might

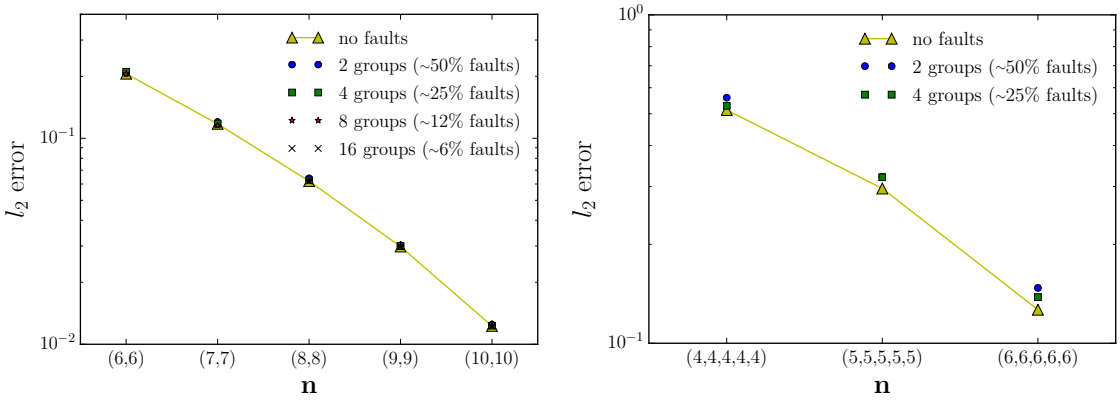

Figure 4.4.: Convergence of the fault-tolerant combination technique applied to the advection-diffusion problem in 2D **(left)** and 5D **(right)**.

be a reason to repeat the whole computation), one has to appreciate the fact that this outcome was obtained in the very extreme case where half of the component grids were lost, without significant computation overhead (which will be demonstrated in the next section).

A one-dimensional projection of the combined solution at the end of the simulation is shown in Figure 4.5. It can be observed that the recovered solution, where one out of four process groups failed, is a very good match to the solution obtained without faults.

4.4.2. Scalability

The overhead of the fault-tolerant combination technique in a massively parallel setup was investigated in scaling experiments on the supercomputer *Hazel Hen*. The experimental setup was a five-dimensional combination technique with $\vec{n} = (8, 8, 7, 7, 7)$ and $\vec{l}_{\min} = (4, 4, 3, 3, 3)$. This resulted in 126 component grids. The experiments were performed with 8192, 16384, 32768 and 65536 processes distributed onto 8, 16 or 32 process groups of size 512, 1024, 2048 or 4096. In all cases, one (simulated) process fault occurred during the second iteration. For computations performed with the same number of process groups, always a process in the same group failed.

The scaling results are presented in Figure 4.6. The figure shows the runtimes for the most significant steps of the fault-tolerant combination technique algorithm: the time to *solve* all component grids for one time step, measured with all process groups, before the fault occurs; the time to *redistribute* the component grids of the failed group; and the time to *recompute* certain component grids if necessary. The other runtimes of the recovery procedure, including the time to solve the GCP, were so low that they are not presented here. The runtime for the combination step was below one second in all cases and thus was also neglected here. The relatively high runtimes for the *solve* step can be explained by a rather bad node-level performance of the application code. However, it scales well, as it can be observed by the almost linear decrease of the runtime when, for a particular number of process groups, the size of the groups is increased.

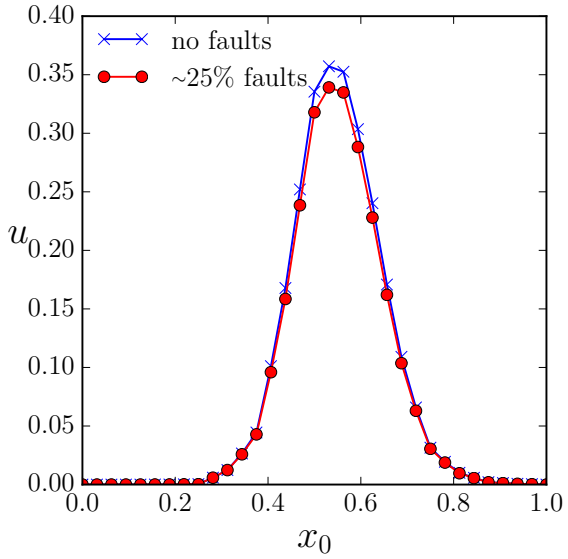

Figure 4.5.: One-dimensional projection of the five-dimensional solution for $\vec{n} = (5, 5, 5, 5, 5)$, with and without faults.

The slightly erratic behavior of the curves can be explained by the following factors. One reason is that there is some degree of randomness in the assignment of the component grids to the process groups, because the assignment depends on the actual time to solve the component grid for the first outer iteration (compare Section 3.2). This means that for the experiments with the same number of process groups, even when always the same group fails, the component grids that are lost can vary for the different sizes of process groups. The results, however, are only shown for a single simulation run. Although a statistical study (e.g., repeating the experiments with different process groups to fail) would have helped to create "smoother" lines in the plot, this would have wasted a substantial amount of compute resources without bringing much further insight. The goal of this study was to identify the most expensive steps of the recovery procedure and to find out whether these steps can scale to a large number of processes. This was possible with the experiments presented here. However, the actual runtimes obtained with the example application are not of interest at all. Thus, no statistical study was performed.

The time for the *redistribute* step was essentially the time for the initialization procedure of the application when the lost component grids are recreated on the healthy process groups. Usually one would expect the time for the redistribution to decrease when more process groups are used, because (when only one group fails) the number of lost component grids decreases. However, this cannot always be seen in the results. As for 16 and 32 process groups the number of component grids to be redistributed was lower than the number of available process groups, here the time was dominated by the grid with the slowest initialization routine. Furthermore, the initialization routine of the example application does not scale very well with the number of processes.

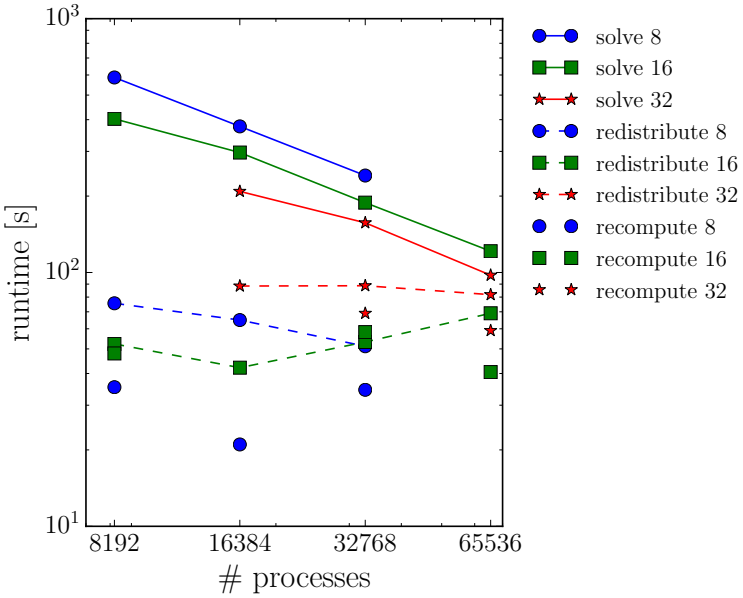

Figure 4.6.: Scaling experiments. The x-axis indicates the total number of processes used for the computations. The numbers in the legend indicate the number of process groups.

After the recovery procedure the time for the *solve* step increases, because less process groups are available. This might become a problem if multiple groups fail in the course of the computation, but can be limited by some of the approaches discussed in Section 4.2.

The results of this study indicate that the overhead of the FTCT is very specific to the particular application and the expected failure rate of the system. It depends on various parameters, such as the expected number of outer iterations that can be computed between two failures, the time for the computation of an outer iteration, and how expensive the initialization routine is compared to a single outer iteration. However, upper bounds for the two most expensive steps of the recovery procedure can easily be formulated:

- The *redistribute* step will never be more expensive than to initialize all component grids.
- The *recompute* step will never take longer than the *solve* step.

This means, that if the time for the initialization of the component grids is small compared to the time for the outer iterations that can be computed in-between two failures, then the overhead of the FTCT is negligible. This is the main conclusion that can be drawn from this study.

5. The combination technique applied to gyrokinetic simulations with GENE

The parallel combination technique was applied to the simulation code GENE [1], which solves the gyrokinetic equations, a five-dimensional, and thus high-dimensional, PDE. It uses regular grids for the discretization and employs distributed memory parallelism based on MPI. It was reported to scale on different large-scale and massively parallel systems [33, 69]. Furthermore, it is part of the "Unified European Applications Benchmark Suite" [5] of PRACE (Partnership for Advanced Computing in Europe), which is used to evaluate the performance of new supercomputers. Thus, GENE is a well-suited candidate to perform experiments with the combination technique in a massively parallel setup and with a sophisticated high-dimensional application. The goal of these experiments is to demonstrate that the parallel combination technique can be successfully applied to initial-value problems in GENE and that this approach has the potential to push the computational limits of gyrokinetic simulations.

First investigation of the combination technique applied to GENE were presented within the project EXAHD by Kowitz [63, 64, 65] and are summarized in his PhD thesis [62]. Although the main focus of his work was the eigenvalue problem [62, 63, 64], he also presented first experiments with linear initial value simulations [62, 65]. However, his experiments with the initial-value problem were limited to a single combination step once in the end of the simulation and to only a relatively short simulation time. Furthermore, he combined the component grids by processing the output files of GENE, after the component grids had been individually computed. As this strategy is rather inefficient, it does not scale to large problem sizes. He concluded from his experiments that the combination technique can be successfully applied to this problem, but that the simulation of a longer simulation time would require a suitable time integration scheme of the form as described in Section 2.6.

With the software framework for the parallel combination technique presented in this work it is now possible to perform experiments with multiple combination steps in an efficient manner, even for large problem sizes. The following experiments demonstrate that frequently combining the component grids can, in fact, significantly improve the accuracy of the combined solution. Furthermore, it is the first time that the accuracy of the combined solution was investigated with experiments that were performed with a sufficient number of time steps so that the simulation time was in a regime where the solution was actually converged. Normalizations are proposed that enable to combine the component solutions once in the end after the individual component solutions have converged. Additionally, the combination technique was also applied to the eigenvalue and to other quantities of interest for linear initial value simulation in GENE. Furthermore, it

is described how to integrate GENE with the parallel combination technique framework in order to enable large scale experiments.

In [6, 9], a parallel combination technique was applied to GENE as well. The main differences of this alternative approach for a parallel combination technique are discussed in Section 3.6. However, the experiments with GENE were only performed with a relatively low number of time steps and thus this only served as a proof of concept. For a meaningful investigation of the approximation quality of the combined solution longer simulation times are required.

5.1. GENE

The gyrokinetic equations are a model to describe the turbulent transport of heat and particles in magnetically confined plasmas [15, 27], which are important issues in the development of future nuclear fusion devices. Although, compared to the full kinetic description, the gyrokinetic equations already reduce the dimensionality of the problem from six to five dimensions and remove irrelevant effects of the smallest temporal scales, the numerical solution of this problem is still very resource intensive. A computational grid required to fully resolve a large-scale fusion experiment like ITER [3] is prohibitively expensive. The combination technique is a promising approach to push the computational frontiers of gyrokinetic simulations, because it enables to obtain such extreme resolution levels with significantly less computational resources. Furthermore, the combination technique might help to ensure the scalability of simulations with GENE on future exascale machines.

The gyrokinetic Vlasov equation

$$\frac{\partial F_s}{\partial t} + \frac{d\vec{X}}{dt} \cdot \nabla F_s + \frac{dv_\parallel}{dt}\frac{\partial F_s}{\partial v_\parallel} + \frac{d\mu}{dt}\frac{\partial F_s}{\partial \mu} = 0 \tag{5.1}$$

describes the propagation of the five-dimensional distribution function $F_s(x, y, z, v_\parallel, \mu)$ of a species s in time. The different species are the different types of particles (typically ions and electrons) that are simulated. GENE employs the so-called δf-splitting, where the distribution function is split into a Maxwellian background distribution $F_{0,s}$ and a fluctuating part g_s. As the detailed representation of g_s is rather involved and not relevant for the following discussions, here the governing equation being solved in GENE can be expressed symbolically as

$$\frac{\partial \vec{g}}{\partial t} = \mathcal{L}\vec{g} + \mathcal{N}\vec{g}, \tag{5.2}$$

where $\vec{g}$ is a vector composed of the different g_s and the mathematical operators $\mathcal{L}$ and $\mathcal{N}$ represent the linear and non-linear part of the equation. For a detailed description of the solution of the gyrokinetic equations in GENE please refer to [71, 34, 22].

GENE offers two main modes of operation, local and global. In local simulations, the domain is a so-called flux-tube whose center follows the magnetic field line. The radial extent of this flux-tube is only small compared to the size of the fusion device and thus

the variation of the temperature and density profiles in radial direction can be neglected. In global simulations, the extent of the domain in radial direction is large and can span the whole machine size. Here, the variation of the temperature and density profiles can not be neglected, and the computational grid is significantly larger than for local simulations. Furthermore, it is possible to approach Equation (5.2) in multiple ways. In order to simulate the plasma turbulence in space and time all terms of the equation are used. However, it is also possible to just consider the linear part of Equation (5.2)

$$\frac{\partial \vec{g}}{\partial t} = \mathcal{L}\vec{g} \tag{5.3}$$

to study the instabilities driving the turbulence. This is done in two ways. Linear initial value simulations are performed to identify the growth rate and frequency of the fastest growing mode. Eigenvalue calculations are used to also find the other unstable modes which are represented by the eigenvalues with positive real part of $\mathcal{L}$.

In this work, local linear initial-value simulations are considered. The purpose of these simulations is to identify the most unstable mode in the plasma, represented by the eigenvalue $\lambda \in \mathbb{C}$ of $\mathcal{L}$ with the largest (positive) real part. The real and imaginary part of λ are denoted as $\gamma = Re\{\lambda\}$ and $\omega = Im\{\lambda\}$. In this work, only simulations with one species are considered and thus, for simplicity, the form

$$\frac{\partial g}{\partial t} = \mathcal{L}g \tag{5.4}$$

is used, where $g \in \mathbb{C}^5$. For $t \to \infty$ the solution will converge to the eigenvector v corresponding to the eigenvalue with the fastest growth rate γ. Thus, when the solution is converged, it holds

$$\frac{\partial g}{\partial t} = \lambda v \tag{5.5}$$

and

$$g(t + \Delta t) = e^{\lambda \Delta t} g(t). \tag{5.6}$$

One can estimate λ by

$$\lambda \approx \frac{\sum_i g_i(t)\lambda_i}{\sum_i g_i(t)} \text{ with } \lambda_i = \ln\left(\frac{g_i(t)}{g_i(t-T)}\right)/\Delta t, \tag{5.7}$$

where g_i are the degrees of freedom of the spatially discretized g and T is an appropriately chosen time interval (in GENE usually 10-20 time steps are used). A practical way to check for convergence is to make sure that the variance of the λ_i is below a certain threshold C

$$\frac{\sum_i |\lambda_i - \lambda|^2 g_i}{\sum_i g_i} < C. \tag{5.8}$$

A very similar convergence criterion is actually used in GENE to check whether the solution has converged with a given precision `omega_prec` [71].

GENE uses a coordinate system aligned to the magnetic field line. Here, x is the coordinate in radial direction and z the coordinate in the direction parallel to the magnetic

field line. The y-direction is perpendicular to the other two. The computational grid in the positional space (x, y, z) is a rectangular mesh with equidistant grid spacing. However, a different mesh width is possible in each direction. The complex curvilinear nature of the coordinate system is only reflected in the computation of the derivatives and integrals of the gyrokinetic equations. In local simulations the simulation domain is periodic in x- and y-direction and thus these directions are represented in Fourier space, as this is beneficial in terms of computational efficiency. The two-dimensional velocity space is composed of the velocity parallel to the magnetic field line $v_\parallel$ and the magnetic moment μ. An equidistant grid is used for the discretization of the $v_\parallel$-direction. In the μ-direction equidistant grids are possible, as well. However, for the numerical integration performed in this direction GENE also offers quadrature rules that employ non-equidistant grids. In z- and $v_\parallel$-direction, Arakawa type finite differences [10] are employed. A fourth-order Runge-Kutta method is used for the time integration. The parallelization in GENE follows the typical domain decomposition approach, as introduced in Chapter 3, where the computational grid is distributed onto different processes that are arranged in a Cartesian process grid. MPI is used for the communication between the processes.

5.1.1. Interface

GENE is adapted to the combination technique software framework by a specialization of the `Task` class (compare Section 3.5), `GeneTask`. The responsibility of this interface is to start GENE (the `solve` function in Algorithm 3) and handle the access to the simulation data which is necessary for the combination step.

Although a true black box approach, where the application code does not need to be modified, would have been highly desirable, this was not possible with GENE. Firstly, the computational grids used by GENE had to be modified in order to make them compatible with the nested grid structure required by the combination technique. These modificiations were implemented by Kowitz [62] for his work and were reused here. Secondly, access to the simulation data has to be provided. In principle, this would have been possible without modifications to the code, by reading and writing the checkpoint files, as it was done by Kowitz [62]. However, for time-dependent problems that require frequent combination this is not an option, because working with files is relatively slow and would not be feasible for large-scale problems. The most efficient way is to access the simulation data directly in the main memory. As this is not intended by the original GENE code, this required further modifications. However, both the modification of the computational grids and the access to the simulation data were achieved without major modifications to the GENE code. Thus, one can speak of a minimal invasive approach.

The access to the simulation data is implemented using the checkpoint mechanism in GENE. After the computation of the current outer iteration is finished, GENE calls the function `checkpoint_write` which would normally write the current simulation data to a file in the form of a so-called checkpoint. The code was modified so that, instead of this function, the function `checkpoint_write_memory` is called, which copies the simulation data into the corresponding instance of the `GeneTask` class. Here, before the combination step, it is then converted to the structure of the component grid, as described

in Section 5.1.2. In the same way, the combined solution is brought back into GENE after the combination step when GENE is started again for the next outer iteration. This is implemented with the function `checkpoint_read_memory`, which replaces the original `checkpoint_read`.

The original functions `checkpoint_write` and `checkpoint_read` use MPI-IO to store the checkpoint to a file. These operations are rather expensive, as they require additional communication and synchronization. Furthermore, the time to write the file is bound by the performance of the parallel file system and does not necessarily scale with the number of processes in the process group. The functions `checkpoint_write_memory` and `checkpoint_read_memory`, however, do only involve copying the data in main memory and thus are are purely local operations which do not require further communication. In comparison to reading and writing the checkpoint file, these operations are very cheap. Furthermore, the runtime of these functions decreases linearly with the number of grid points per process and thus with the size of the process group.

5.1.2. Alignment of grids and boundary conditions

The combination technique requires component grids that are nested, i.e., the grid points of discretization level l are included in the grid with discretization level $l+1$, or higher. This is usually achieved by using equidistant grid spacing with $2^{l_k} \pm 1$ grid points in each dimension k.

For simulations with GENE, however, it is more common to use even numbers of grid points, as this enables more flexibility to choose the parallelization, i.e., the number of processes used in each direction. The reason is that GENE requires that the number of grid points in each direction divides by the number of processes in this direction. Thus, with an odd number of grid points it would be rather difficult to find a parallelization so that all component grids can be computed with the same number of processes. However, when the number of grid points and the number of processes in each direction is a power of two, this is easily possible. Thus, for the experiments in this work, 2^{l_k} grid points in each direction were used in GENE. This requires, however, that the grids in GENE have to be modified so that they are compatible with the combination technique. These modifications were proposed in [62]. In this work, they were extended to allow for a massively parallel execution.

Fourier directions k_x, k_y

For the local linear initial value computations in GENE, the x- and y-direction are formulated in Fourier space. The coordinates for the discretization of the frequency space, k_x and k_y, correspond to a certain frequency or wave number and the coefficient assigned to each such grid point is a coefficient of the Fourier representation. A spatial discretization with $2^l + 1$ grid points in the, say, x-direction results in a (discrete) Fourier representation with $2^l + 1$ different wave numbers in the k_x-direction. On grids with a high resolution, a wider range of wave numbers is represented than for grids with a low resolution. This is visualized in Figure 5.1. The projection (interpolation) of a grid

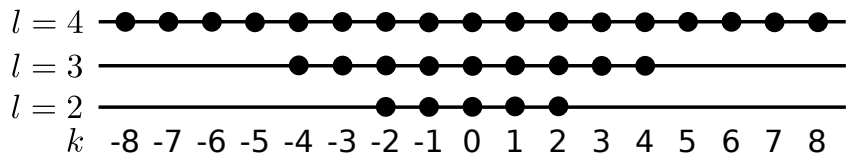

Figure 5.1.: Alignment of grid points for different discretizations of the frequency space. The numbers are the indices corresponding to the different wave numbers. Adapted from [62].

formulated in the Fourier representation into a grid with higher resolution is achieved by setting the Fourier coefficients to zero that exceed the wave number range of the grid with low resolution. The combination technique is then straight-forward

$$u_{\mathcal{I}}^{(c)}(\vec{k}) = \sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} \tilde{u}_{\vec{l}}(\vec{k}), \tag{5.9}$$

where

$$\tilde{u}_{\vec{l}}(\vec{k}) = \begin{cases} u_{\vec{l}}(\vec{k}) & \text{if } \vec{k} \in K_{\vec{l}} \\ 0 & \text{else.} \end{cases} \tag{5.10}$$

Here, $K_{\vec{l}} = \left\{ \vec{k} \in \mathbb{Z}^d : -2^{l_i} \leq k_i \leq 2^{l_i}, 1 \leq i \leq d \right\}$ denotes the index set of wave numbers for each grid. An investigation of the Fourier transformation with sparse grids can be found in [41, 35, 37].

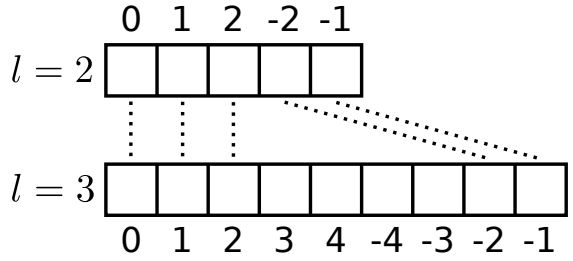

Figure 5.2.: Storage order of Fourier directions k_x and k_y. The numbers are the indices corresponding to the different wave numbers.

Although the concept of combining component grids in the Fourier dimensions is straight-forward, there are some technical issues that need to be considered for the combination step. First of all, in GENE the different wave numbers are not stored in the order as presented in Figure 5.1. GENE follows the index ordering of the FFTW library [26], which is used for the discrete Fourier transformation. As visualized in Figure 5.2, first all the positive indices are listed and then the negative ones. This order has to be considered in the algorithms for the combination step, when adding up grids of different discretization levels. Furthermore, no hierarchization is used in these coordinate directions. This can easily be achieved by ignoring the respective directions in the uni-directional hierarchization algorithm (see Section 2.3).

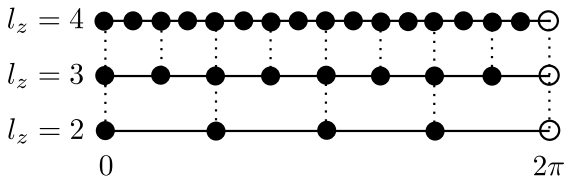

Figure 5.3.: The black dots indicate the GENE grid in z-direction for different discretization levels. The circles indicate the additional boundary points of the component grid. Reprinted from [62].

Although parallelization of the Fourier directions is possible in GENE, and in FFTW, it is usually avoided. The reason is that FFTW does not scale very well when executed in a distributed memory fashion, as this involves significant communication overhead. Thus, for the experiments in this work, parallelization of the Fourier directions was not considered.

z-direction

In the parallel direction z, GENE employs a special kind of periodic boundary condition. This boundary condition has to be represented on the component grid in order to allow for a proper evaluation of the interpolant close to the boundary.

In GENE, the grid points are placed equidistantly in the interval $[0, 2\pi - \Delta z]$, where Δz is the grid spacing. As visualized in Figure 5.3, using 2^{l_z} grid points in this direction results in nested grids, which is favorable for the combination technique. Due to the periodicity, a grid point at $z = 2\pi$ would be redundant and thus is not explicitly stored by GENE.

In order to make the GENE grid compatible with the structure of the component grid, the function value at $z = 2\pi$ has to be provided. Otherwise, the evaluation of the component solution in the interval $[2\pi - \Delta z, 2\pi]$ would not return the right result. The periodicity in z-direction is described as

$$g(k_x, k_y, z + 2\pi, v_\parallel, \mu) = g(k'_x, k_y, z, v_\parallel, \mu)(-1)^{Nl}, \tag{5.11}$$

where $k'_x = k_x^{\min}(j + Nl)$. Here, $j, l \geq 1$ denote the indices corresponding to the wave numbers, i.e., $k_x = k_x^{\min} j$ and $k_y = k_y^{\min} l$. The values $k_x^{\min}$, $k_y^{\min}$ and $N \in \mathbb{N}$ are parameters specific to the simulation scenario at hand, with

$$N = \hat{s} k_y^{\min} L_x, \tag{5.12}$$

where $\hat{s}$ is the so-called safety factor and L_x the size of the simulation domain in x-direction. For a detailed derivation of the boundary condition in z-direction please refer to [62].

In the case that there is no parallelization in z-direction, setting the boundary points is straightforward. After the data has been copied from GENE into the component grid, one iterates over the component grid in all other directions than the z-direction, and sets

the right-hand boundary of z according to Equation (5.11). Here, it can happen that $k_x' > k_x^{\max}$, where $k_x^{\max}$ is the highest wave number. In this case $g(k_x, k_y, z+2\pi, v_\parallel, \mu) = 0$. This is in agreement with the numerical treatment in GENE, where a Dirichlet zero boundary is enforced in k_x-direction under the assumption that the Fourier coefficients decay to zero for $k_x \to \infty$.

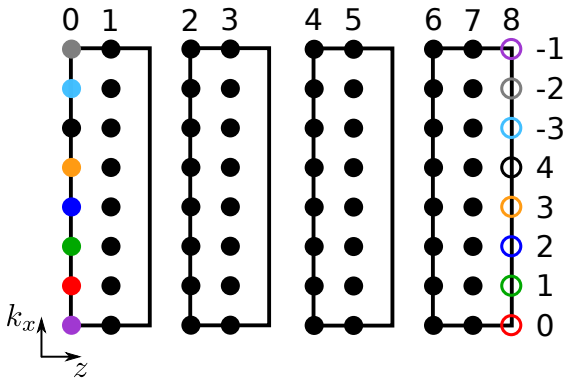

Figure 5.4.: Adaption of the z-boundary on a distributed component grid with four processes in z-direction. The dots indicate the grid points of the GENE grid. The circles indicate the additional boundary points of the component grid. The colors indicate how the grid points are connected through the boundary condition (for the case $Nl = 1$ in Equation (5.11)).

When parallelization is used in z-direction, data has to be exchanged between the first and last process in z-direction. This is visualized in Figure 5.4. Note that the ordering of the indices in k_x-direction follows the FFTW notation. The grid points marked with the same color are connected by the parallel boundary condition (see Equation (5.11)). This means, the grid points with coordinate $z = 0$ have to be sent from the first process in z-direction to the last process. This is implemented using `MPI_Subarray` (to specify a $(d-1)$-dimensional slice of the component grid on the sending side) and a temporary $(d-1)$-dimensional array on the receiving side to buffer the data before copying it to the correct position in the component grid. Note that the amount of data that is transferred does not scale with the number of processes in z-direction and thus very anisotropic domain decompositions towards this direction should be avoided. In this work, no parallelization was used in the k_x- and k_y-direction for the reasons explained above.

$v_\parallel$-direction

In the $v_\parallel$-direction, GENE uses an equidistant grid spacing in the interval $\left[-v_\parallel^{\max}, v_\parallel^{\max}\right]$. Choosing $2^{l_{v_\parallel}}$ grid points in this direction results in grids that are not nested for different discretization levels (compare Figure 5.5 (left)). In order to obtain nested grids with $2^{l_{v_\parallel}}$ grid points, Kowitz [62] slightly modified the discretization of the $v_\parallel$-direction in GENE.

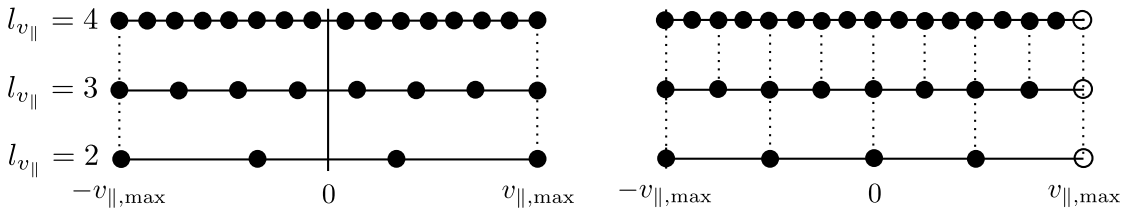

Figure 5.5.: **Left:** Original discretization of the $v_\parallel$-direction in GENE for different discretization levels. **Right:** The black dots indicate the modified GENE grid. The circles indicate the additional grid points at the boundary that are added when converting the GENE grid to the structure of the component grid. Adapted from [62].

This is visualized in Figure 5.5 (right). The grid spacing has been changed from

$$\Delta v_\parallel = \frac{2 v_\parallel^{\max}}{2^{l_{v_\parallel}} - 1} \tag{5.13}$$

to

$$\Delta v_\parallel = \frac{2 v_\parallel^{\max}}{2^{l_{v_\parallel}}}. \tag{5.14}$$

Kowitz further made sure that the modified discretization does not have adverse effects on the convergence behavior. For details on this issue, please refer to [62].

In order to make the modified GENE grid compatible with the structure of the component grids, as visualized in Figure 5.5, a grid point is added on the right-hand boundary. The function value corresponding to this grid point is set to zero. As the solution g decays to zero for $v_\parallel \to \pm\infty$, the additional approximation error is negligible.

μ-direction

For the μ-direction different grid layouts are available in GENE, corresponding to the different quadrature rules that are available. GENE offers Gauss-Legendre, Gauss-Laguerre and the trapezoidal rule as quadrature rules for the numerical treatment of this direction. Although the trapezoidal rule has the slowest convergence rate of these three quadrature rules, it is the only one that uses equidistant grid spacing and thus can provide the required nested grid structure.

The grid originally used in GENE for the trapezoidal rule consists of equidistantly distributed grid points in the interval $[0, \mu_{\max}]$, with coordinates

$$\mu_i = \Delta\mu(i - 0.5) \quad \text{for } i = 1, 2, \ldots, n_\mu, \tag{5.15}$$

where n_μ is the number of grid points in this direction and $\Delta\mu = \mu_{\max}/n_\mu$. As visualized in Figure 5.6 (left), this grid is not nested for different discretization levels and thus not suitable for the combination technique. For this reason, Kowitz [62] introduced a modified version of this grid. Slightly shifting the grid points

$$\mu_i = \Delta\mu(i - 1) \quad \text{for } i = 1, 2, \ldots, n_\mu, \tag{5.16}$$

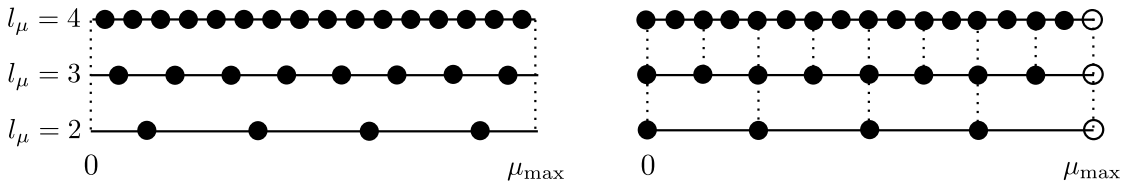

Figure 5.6.: **Left:** Original discretization of the μ-direction in GENE for different discretization levels. **Right:** The black dots indicate the modified GENE grid. The circles indicate the additional grid points at the boundary that are added when converting the GENE grid to the structure of the component grid. Adapted from [62]

creates nested grids for $n_\mu = 2^{l_\mu}$. Similar as in the case of the $v_\parallel$-direction, there is an additional grid point on the right-hand boundary of the component grid that corresponds to the GENE grid. As g decays to zero for $\mu \to \infty$, again the function values corresponding to these additional grid points can be set to zero without introducing a noticeable additional error.

In [62], a quadrature rule for the μ-direction in GENE based on Clenshaw Curtis quadrature is proposed. With this quadrature rule the convergence rate can be significantly improved, while preserving the nested grid structure. However, an implementation of the distributed combination step that supports component grids that have the Clenshaw Curtis grid layout is not yet available. This would be an interesting topic for future work.

Multiple species

Although for the experiments in this work only one species was used, here it is briefly addressed how the combination technique can be applied to simulation scenarios with more than one species. When multiple species are used in GENE, they are stored as an additional dimension of the multidimensional array that is used on each process to store the simulation data.

There are now two possible ways to apply the combination technique to simulation scenarios with multiple species. One can either store an individual component grid for each species, or use the array structure of GENE and virtually increase the dimensionality of the component grids to six dimensions. In the latter case, the additional dimension is then ignored in the algorithms for the hierarchization and dehierarchization. In both cases, all available data structures and algorithms can be reused. Independent of the chosen implementation, the storage size and the time for the combination step increases linearly with the number of species.

5.2. Experiments and results

The simulation scenario used in the experiments is the local linear initial-value simulation of a so-called ion-temperature-gradient instability (ITG), where an instability is induced by the gradient of the ion temperature. It is the same application scenario that has

also been used by Kowitz [62, 65]. The corresponding parameters can be found in Appendix B.1. The computations were performed on the supercomputer *Hazel Hen* (see Appendix A).

In the following, different three-dimensional combination techniques applied to the eigenvalue λ, to the eigenvector g and other quantities of interest in GENE were investigated. The dimensions of the gyrokinetic equations considered for the combination technique were z, $v_{||}$ and μ. This means, that a fixed resolution was used in the spectral dimensions k_x and k_y for all component grids.

From the investigations in Kowitz [62] one can conclude that it is not yet clear whether considering the k_x-direction for the combination technique makes sense (when relating the loss in accuracy to the saved computational resources). Thus, in order to avoid unnecessary complexity for these first experiments with the parallel combination technique, the k_x dimension was not considered for the combination technique. For this simulation scenario it can be observed that at convergence, for both full grid and combined solution, the solution is non-zero only for the inner wave numbers in k_x-direction with the indices $-1, 0, 1$. This can be observed for any choice of $l_x \geq 3$. Therefore, one would expect that using a higher target resolution in k_x-direction than $l_x = 3$ does not improve the accuracy of the solution. Nevertheless, for other, more complex simulation scenarios which require high resolutions in the k_x-direction, there might be a real (performance) benefit of applying the combination technique in this direction. However, further investigations of this issue were not in the scope of this work.

For local linear simulations the system decouples in the k_y-direction and thus each k_y-coordinate can be treated in an individual simulation run with only one grid point in this direction. Thus, considering this dimension for the combination technique does not have any advantage. This means the discretization levels of the dimensions k_x and k_y are fixed and only the remaining three dimensions z, $v_{||}$ and μ are considered for the combination technique.

Table 5.1.: List of combination technique experiments.

	$\vec{n}$	$\vec{l}_{\min}$	# component grids
combi 4 grids	(3,1,5,5,5)	(3,1,4,4,4)	4
	(3,1,6,6,6)	(3,1,5,5,5)	4
	(3,1,7,7,7)	(3,1,6,6,6)	4
	(3,1,8,8,8)	(3,1,7,7,7)	4
combi 10 grids	(3,1,6,6,6)	(3,1,4,4,4)	10
	(3,1,7,7,7)	(3,1,5,5,5)	10
	(3,1,8,8,8)	(3,1,6,6,6)	10
combi $\vec{l}_{\min} = (3,1,4,4,4)$	(3,1,7,7,7)	(3,1,4,4,4)	19
	(3,1,8,8,8)	(3,1,4,4,4)	31

The different combination techniques are:

- *combi 4 grids.* A combination technique with 4 grids. Here, the distance between target level and minimum level is only one (in each of the three dimensions). This

should produce the best accuracy, but is theoretically (number of grid points) more expensive than the corresponding full grid solution.

- *combi 10 grids*. A combination technique with 10 grids. The distance between target level and minimum level is two (in each of the three dimensions)
- *combi* $\vec{l}_{min} = (3, 1, 4, 4, 4)$. The minimum level is constant, i.e. the number of grids increases when $\vec{n}$ is increased. This is the combination technique in the classical sense.

Table 5.1 lists the experiments. Note that the first experiment of *combi 4 grids* and the first of *combi 10 grids* would also fit in *combi* $\vec{l}_{min} = (3, 1, 4, 4, 4)$. However, each experiment is only assigned to one group of experiments to avoid confusion.

The lowest minimum level used in any of the experiments was adjusted so that GENE converges for all component grids with the chosen stopping criterion `omega_prec` $= 10^{-6}$. In order to test this, all component grids were computed individually. Furthermore, an equal minimum level in all three dimensions considered for the combination technique was used to retain a symmetric combination scheme. These two conditions resulted in the choice (3,1,4,4,4).

One can speak of a successful application of the combination technique to a particular problem, if the combined solution converges (to the true solution) when increasing the target level and if the approximation on each component grid is worse than the combined solution. Since for many problems the true solution is not known, the former can be checked by comparing the combined solution to a sufficiently resolved full grid reference solution, which is known to converge to the true solution. The latter can be checked by comparing the component solutions to this reference solution as well. If there is a component solution which is better than the combined solution, this would either mean that the combination technique does not work for this problem at all (e.g., because some of the numerical preconditions are not fulfilled), or that the combination scheme was not appropriately chosen (e.g., some of the most anisotropic component grids have a significantly worse approximation quality than the other component grids and thus destroy the combined solution). Another, more economic, question is whether the application of the combination technique is reasonable with respect to the use of computational resources: can the combination technique produce a solution of comparable accuracy at lower computational cost? The goal of the following experiments is to answer these questions.

5.2.1. Eigenvalue λ

The eigenvalue λ is one of the most important results of linear initial value simulations with GENE. The combination technique can be directly applied to these scalar values

$$\lambda_{\mathcal{I}}^{(c)} = \sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} \lambda_{\vec{l}}. \tag{5.17}$$

Each component solution was computed individually until convergence and the resulting $\lambda_{\vec{l}}$ was extracted from the output file `omega.dat`. The convergence criterion `omega_prec`

was set to 10^{-6}. The time step size was automatically set by GENE (`calc_dt = T`). Hence, each component solution was computed with a different time step size and different number of time steps. As the time step size is not always properly detected in GENE for very anisotropic discretizations, it was further made sure that the simulations do not yield incorrect results due to wrongly set time step sizes by repeating all experiments with a significantly lower time step size.

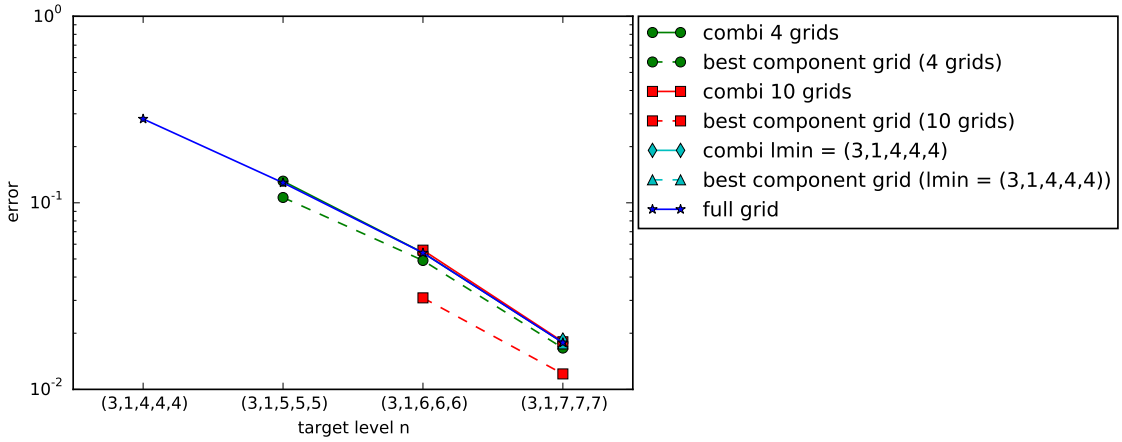

Figure 5.7.: Error of λ compared to the full grid solution with $\vec{l} = (3, 1, 8, 8, 8)$.

Figure 5.7 shows the error for the different experiments compared to the full grid reference solution λ_{ref} computed with $\vec{n} = (3, 1, 8, 8, 8)$. Furthermore, the errors of the full grid solutions with $\vec{n} = (3, 1, 4, 4, 4), (3, 1, 5, 5, 5), (3, 1, 6, 6, 6), (3, 1, 7, 7, 7)$ are shown. The error is defined as $e_{\mathcal{I}}^{(c)} := |\lambda_{\mathcal{I}}^{(c)} - \lambda_{\text{ref}}|$. For each experiment also the error of the best component solution is shown. One can observe that the combined solutions converge to the reference solution and furthermore are very close to the respective full grid solutions. It is noticeable that there are component solutions which are better than the respective combined solutions. What these component grids have in common is an anisotropy in the μ-direction. However, as they are even better than the corresponding full grid solutions, this indicates that an anisotropic discretization in μ-direction is favorable for this problem in general, rather than that the combination technique cannot be successfully applied. With the adaptive combination technique (see Section 2.5) it could be possible to systematically discover such effects.

Figure 5.8 shows the error of the combined solution to the corresponding full grid solution. This means, a combination technique with target level $\vec{n} = (3, 1, 5, 5, 5)$ is compared to the full grid solution with target level $\vec{n} = (3, 1, 5, 5, 5)$, and so on. One can observe that the difference between combined solution and full grid solution becomes smaller with increasing target level and thus one can conclude that the combined solution converges (if both combined solution and full grid solution converge to the true solution, they will eventually approach each other). Furthermore, here, in each case the combined solution is better than the corresponding best component grid. Thus the combined solution is the best approximation (with respect to the component grids used in the

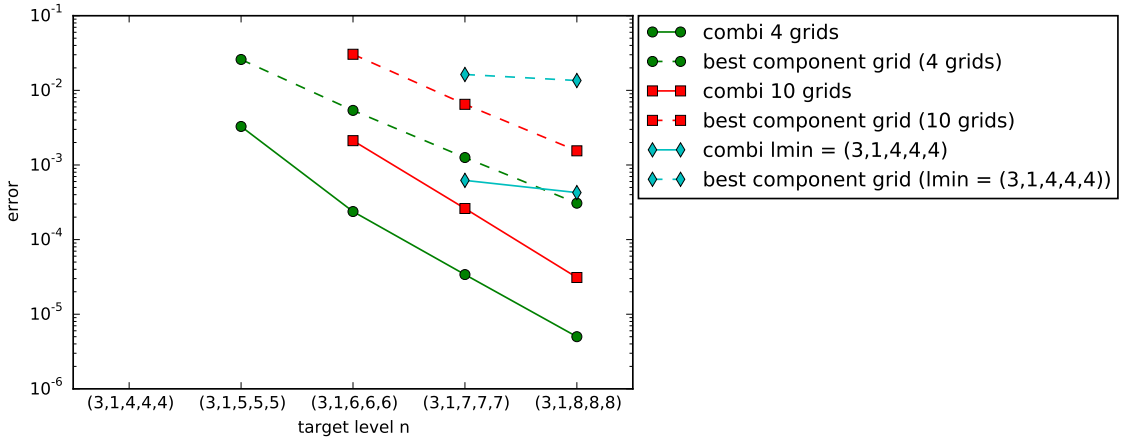

Figure 5.8.: Error of λ compared to the full grid solution with the corresponding target level $\vec{n}$.

combination scheme) to the respective full grid solution, but it is not necessarily the best approximation to a higher resolved full grid solution (as observed in Figure 5.7). One can also observe that the convergence rate of the combination technique with the fixed minimum level is significantly slower than for the other two. This is, however, not surprising if one considers that also the total number of grid points that are computed (summed over all component grids) grows significantly slower in this case when increasing the target level. Whereas in the cases with a constant number of component grids (*combi 4 grids* and *combi 10 grids*) the total number of grid points grows proportionally to the size of the respective full grids, in the case of a fixed minimum level it grows significantly slower than on the full grid.

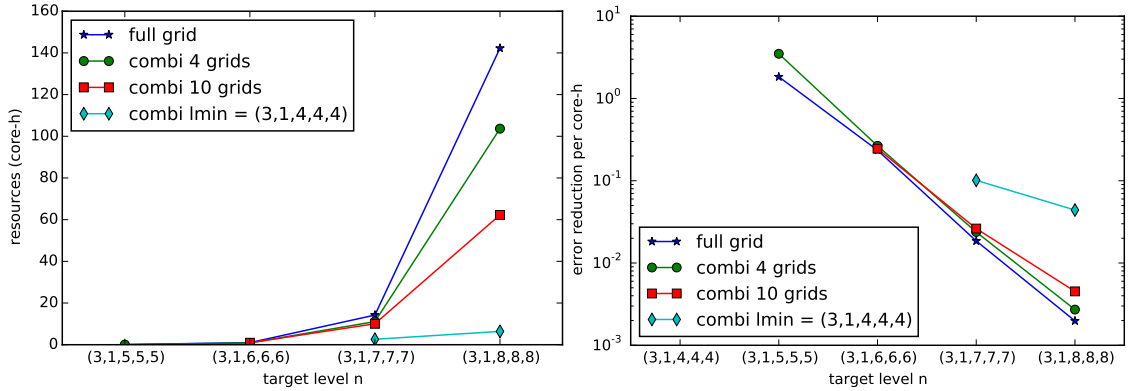

Figure 5.9.: **Left:** Computational resources for the different experiments. **Right:** Error reduction per core hour.

Figure 5.9 (left) shows the computational resources that were spent for the different experiments. The resources are expressed in core hours (core-h), i.e., the runtime of

the computation multiplied with the number of compute cores that were used, in order to make computations with different numbers of processes comparable to each other. One can observe that the resources for the combination technique with a fixed minimum level are significantly lower than for the full grid and also grow significantly slower when increasing the target level. It is noticeable that also the resources for the combination techniques with a fixed number of component grids grow slower than for the full grid, although the computed number of grid points increases proportionally. For actual computations there are, however, other factors that have an impact on the computational cost than just the size of the grids and that are in favor for the combination technique. One important factor is that usually the computational resources grow more than linearly with the problem size, especially in the case of parallel simulations. In this context, it is beneficial to compute several smaller grids than a single large one, even if the total number of grid points is equal. Another advantage of the smaller grids is that also the time step size increases with increasing resolution (at least in the case of explicit time stepping, as it is used in GENE). Thus, using a larger time step size for the combination technique (in the best case the optimal individual time step size is used for each grid) than for the full grid further reduces the computational resources.

The amount of computational resources alone does not allow for a fair comparison of the different experiments, as they all have different approximation quality. In Figure 5.9 (right) the computational resources were related to the error reduction compared to a certain baseline. As baseline e_{base} the error of the full grid solution with $\vec{l} = (3, 1, 4, 4, 4)$ and the corresponding resources as r_{base} are used. Hence,

$$\frac{|e_{\text{base}} - e_{\vec{n}}|}{|r_{\text{base}} - r_{\vec{n}}|} \tag{5.18}$$

denotes the error reduction per core hour in relation to the baseline for a given combined solution or full grid solution with target level $\vec{n}$. The reference for the error calculation was the full grid solution with $\vec{n} = (3, 1, 8, 8, 8)$. One can observe that the error reduction per core hour gets smaller with increasing target level in all cases. Thus, in all cases the convergence rate is slower than the growth of the computational resources. However, the combination technique uses the computational resources more efficiently. Especially, for the combination technique with a fixed minimum level the difference to the full grid is significant (up to one order of magnitude).

5.2.2. Eigenvector g

The combined solution for the eigenvector g is computed as described by Equation (2.23)

$$g^{(c)} = \sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} g_{\vec{l}}. \tag{5.19}$$

However, over time the component solutions $g_{\vec{l}}$ grow exponentially and slight differences in the growth rate (real part of λ) result in severely different magnitudes of the component solutions. Thus, if all component grids are computed individually of each other until

convergence without intermediate combination, the combined solution will eventually converge to the component solution which has the highest magnitude.

This effect can be avoided by frequently combining the component grids with an appropriately chosen outer time step size. This will be demonstrated in the following and the effect of the outer time step size on the accuracy of the combined solution will be investigated.

Another approach to mitigate this problem is to normalize the component solutions before the combination. In the following suitable normalization strategies are proposed and the effect on the accuracy of the combined solution is investigated. It was found that normalization of the component grids can significantly improve the combined solution when the component grids are only combined once in the end. Nevertheless, the best possible accuracy can only be achieved by combining after each time step.

In all experiments the component grids were computed with an equal time step size $\Delta t = 0.005$. This time step size was chosen so that it fulfills the time step size limit of all component grids and of all reference full grids. In all cases, 6000 time steps were computed in total. It was made sure that this number of time steps is high enough so that all individual component grids (and also the full grids) are actually converged with the chosen precision `omega_prec` $= 10^{-6}$. The experiments were repeated with different outer time step sizes and different normalizations.

In order to compute the error of the combined solution, it is interpolated into the function space of the reference full grid. The same is done to compute the error of a full grid to the reference full grid. However, before the error can then be computed in the l_2 vector norm, the solutions need to be normalized. As each discretization results in a (slightly) different eigenvalue λ, all the solutions, combined solution and full grid solution, end up with totally different magnitudes and are also shifted in phase. This can easily be seen when one compares a single solution g at two different points in time, t and $t + T$ after it is converged. When the solution is converged, it holds

$$g(t + T) = g(t) \cdot e^{T\lambda} = g(t) \cdot e^{T(\gamma+\omega i)}, \tag{5.20}$$

where i is the imaginary unit. A suitable approach for the error calculation is to normalize the magnitude of the solutions and to avoid the phase shift by computing the error for the point-wise absolute value of $\tilde{g}$ ($\tilde{g}$ is g interpolated to the grid of the reference solution g_{ref})

$$e_{\tilde{g}} = \sqrt{\sum_j \left(\frac{|\tilde{g}_j|}{\|\tilde{g}\|_2} - \frac{|g_{\text{ref},j}|}{\|g_{\text{ref}}\|_2} \right)^2}, \tag{5.21}$$

where j the denotes the j-th entry in the vectors $\tilde{g}$ and g_{ref}. In Equation (5.20) it can easily be seen that such a normalization removes the exponential growth and phase shift that a solution experiences over time

$$\frac{|g_j(t + T)|}{\|g(t + T)\|_2} = \frac{|g_j(t) \cdot e^{T\gamma} \cdot e^{T\omega i}|}{\|g(t) \cdot e^{T\gamma} \cdot e^{T\omega i}\|_2} = \frac{|g_j(t)| \cdot e^{T\gamma} \cdot 1}{\|g(t)\|_2 \cdot e^{T\gamma} \cdot 1} = \frac{|g_j(t)|}{\|g(t)\|_2}. \tag{5.22}$$

Instead of using the point-wise absolute value for the error calculation, one could also normalize the phase as it is described later. However, for the purpose of a convergence study it does not matter which of the two variants is used.

Linear interpolation was used to interpolate a full grid solution to the resolution of the reference full grid. However, it was made sure with a simple test that this did not deteriorate the convergence rate. When GENE reads in a `checkpoint` file having a lower resolution than the current setup, it uses third-order Lagrange interpolation (in the non-spectral dimensions) to bring the checkpoint to the target resolution. Computing only one time step with essentially zero time step size (e.g., 1e-12), GENE outputs a (nearly) identical checkpoint of higher resolution. This procedure was applied to interpolate all the full grid solutions to the resolution of the reference full grid in order to check whether this results in a different approximation error. However, no noticeable difference between higher-order interpolation and linear interpolation was observed for the error. How the combined solution was evaluated on the reference full grid is described in Section 3.4.

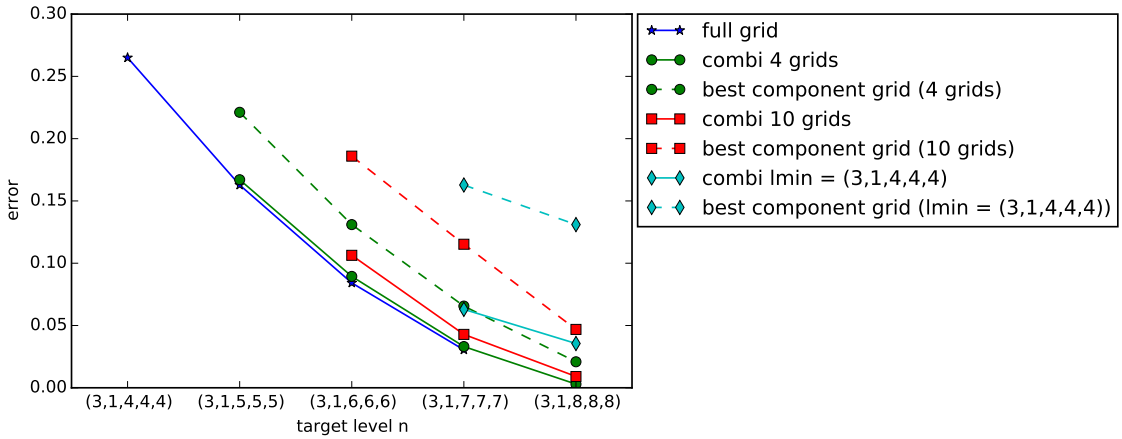

Figure 5.10.: Error of g compared to the full grid solution with $\vec{l} = (3, 1, 8, 8, 8)$. The combined solutions were obtained by combining after each time step.

Figure 5.10 shows the error for the different experiments to the reference full grid with $\vec{n} = (3, 1, 8, 8, 8)$ for the combination after each time step, i.e., $\Delta T = \Delta t$. One can observe that all combination techniques converge to the reference solution. Furthermore, in all cases the combined solution is better than the best component grid. Thus, one can conclude that the combination technique can be successfully applied to the eigenvector g. The accuracy of the different combination techniques in comparison to each other behaves as expected: the finer the underlying sparse grid (and thus the higher the number of total computed grid points), the better the approximation quality. Figure 5.11 shows the error of the combined solutions to the respective full grid solutions. Here, the same trends can be observed.

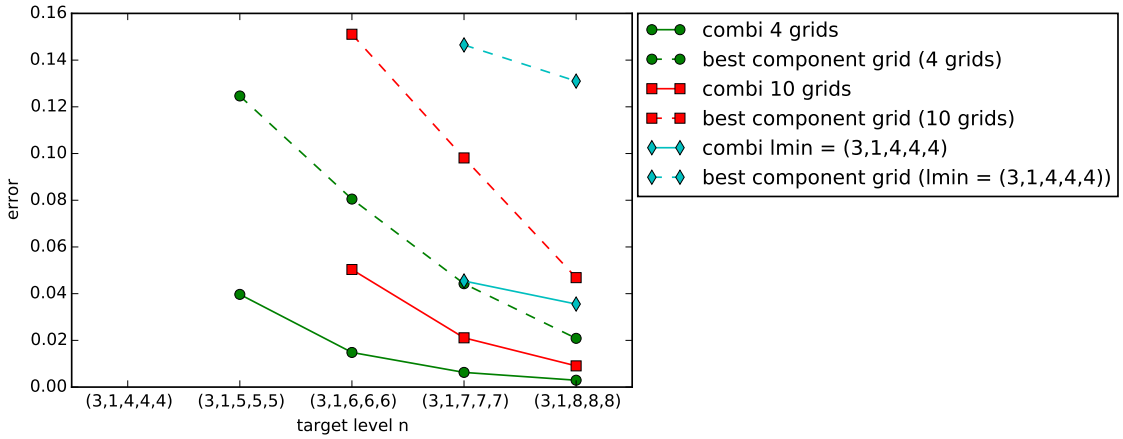

Figure 5.11.: Error of g compared to the full grid solution with the corresponding target level $\vec{n}$. The combined solutions were obtained by combining after each time step.

Influence of outer time step size on approximation quality

Figure 5.12 shows the error of the experiments for different outer time step sizes. All computations were performed with a total of 6000 inner time steps and $\Delta t = 0.005$, as before, but with different outer time step sizes, i.e., 1, 5, 10, 50, 100, 500, 1000 and 6000 inner time steps. An outer time step size of 1 means combining after each inner time step. An outer time step size of 6000 means combining once in the end.

It can be seen that combining once in the end is significantly worse than combining after each inner time step. When combining only once in the end, the combined solution converges to the most dominant component solution. The significant difference in accuracy between an outer time step size of 1000 and 6000 indicates that this effect can already be avoided by only a few intermediate combination steps. However, in-between these two points the error does not increase monotonously, but there are certain outer time step sizes that result in very bad accuracy. The numerical reasons for this behavior are not yet understood and require further investigations. In the following section, it can be observed that in some cases the normalization of the phase mitigates this effect significantly.

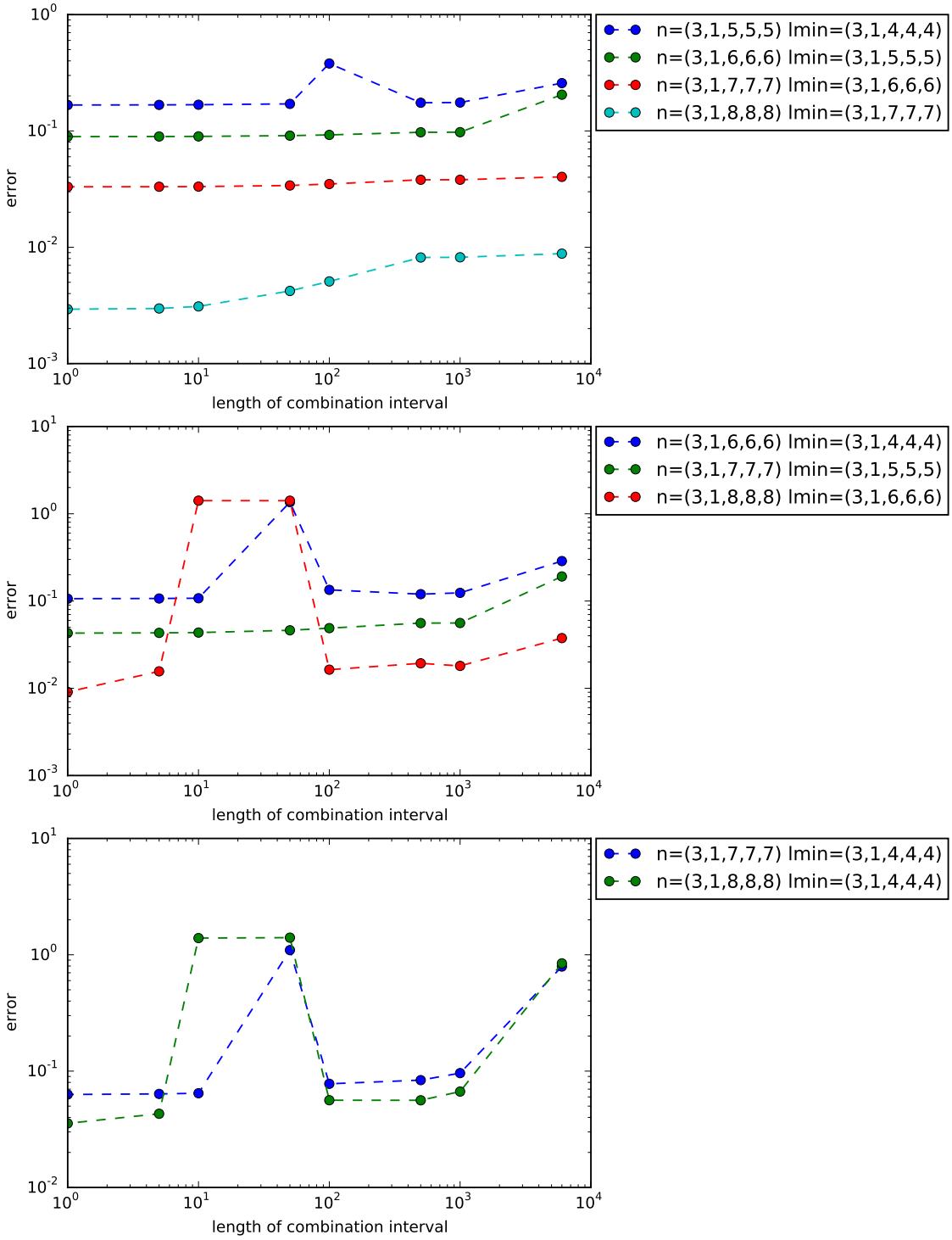

Figure 5.12.: Error of g for different outer time step sizes for *combi 4 grids* (top), *combi 10 grids* (middle) and *combi* $\vec{l}_{min} = (3, 1, 4, 4, 4)$. The x-axis is the number of inner time steps computed during one outer time step.

Influence of normalizations on the approximation quality

Observing the different growth rates and phase shifts that the component solutions experience, led to the idea to normalize the component solutions before the combination step. In the following it is investigated whether a normalization of the amplitude and/or the phase before the combination step can improve the accuracy of the combined solution.

Normalizing the phase means turning the average phase ϕ of a solution vector g (with entries g_j) to zero:

$$\phi = \arg\left(\sum_j g_j\right)$$
$$\tilde{g} = ge^{-\phi i}$$

In principle the amplitude can be normalized in the same way as it is done for the error calculation (see Equation (5.21) and (5.22)). In general, a suitable norm would be the $\mathcal{L}^2$-norm

$$\|f\|_{\mathcal{L}^2} := \left(\int_\Omega |f(\vec{x})|^2 dx_1 \dots dx_d\right)^{(1/2)}, \tag{5.23}$$

where $f(\vec{x}) : \Omega \mapsto \mathbb{R}$ and $\Omega \in \mathbb{R}^d$, or any sufficiently good approximation of it. The l_2 vector norm, however, is not suitable to normalize the individual component solutions, because it heavily depends on the underlying discretization. For example, the same function projected to two different grids, where one grid has twice as many grid points as the other, would result in the norm of the finer grid being roughly twice as high. Fortunately, a quantity that can be used for the normalization is already available as an output of GENE. The square-root of the averaged squared particle density $\tilde{q_0}$ increases with the same rate as g and is independent of the discretization. Thus, it can be used for this purpose and saves additional effort to approximate the $\mathcal{L}^2$-norm by, e.g., quadrature. The normalization of the amplitude is then

$$\tilde{g} = g/\sqrt{\tilde{q_0}}. \tag{5.24}$$

Further details on $\tilde{q_0}$ are presented in Section 5.2.3. These normalizations are then applied to the component grids before the combination step, such that all component grids have the same average phase and/or the same amplitude with respect to $\tilde{q_0}$.

The experiments were repeated for all the different outer time step sizes using different combinations of these two normalizations before the combination step. Figure 5.13 shows the error of the combined solution to the reference solution with $\vec{n} = (3, 1, 8, 8, 8)$ without normalization, with normalization of only the phase, with normalization of only the amplitude and with both normalization of phase and amplitude. It can be seen that in most cases the normalization of phase and amplitude significantly reduces the error when the component solutions are only combined once in the end of the simulation. In all cases the best accuracy was obtained when combining after each time step without normalization. Using normalizations for the combination after each time step resulted in a significantly higher error. In-between the two border cases, combining after each time

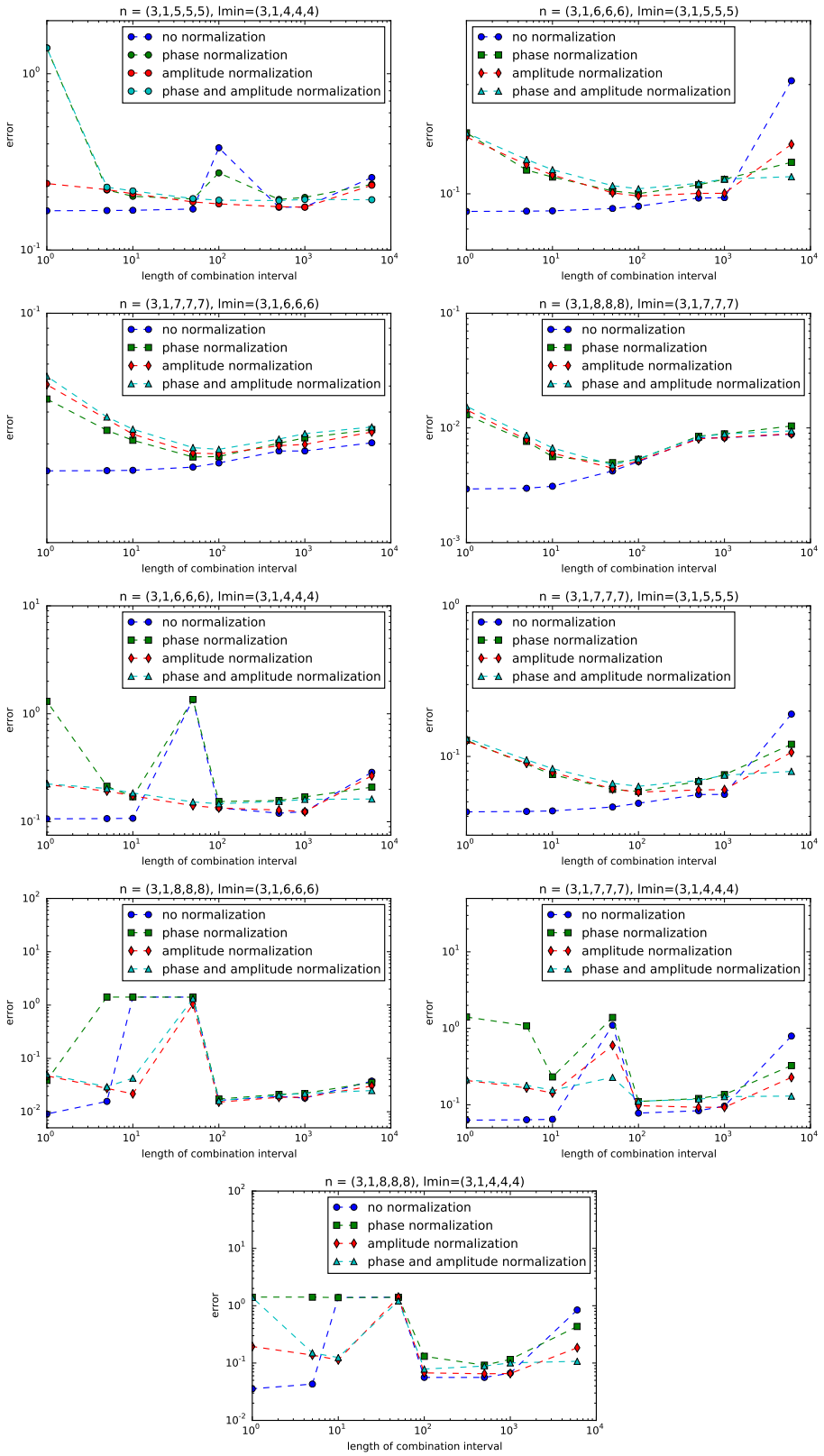

Figure 5.13.: Error of g for different normalizations.

step and combining once in the end, in most cases the normalizations had an adverse effect on the accuracy. However, there are some cases where the solution gets destroyed for certain sizes of outer iterations without normalization. Although in most of these cases normalization of the amplitude could prevent this effect, the accuracy of the combined solution obtained this way is still worse than (without normalization) for a larger outer time step size. This can, for example, be seen for $\vec{n} = (3, 1, 6, 6, 6)$ and $\vec{l}_{\min} = (3, 1, 4, 4, 4)$ which breaks for an outer time step size of 50 iterations. The error for an outer time step size of 500 iterations without normalization is still better here than with amplitude normalization for an outer time step size of 50. Furthermore, there are cases where the normalization of the amplitude did not prevent that the solution breaks.

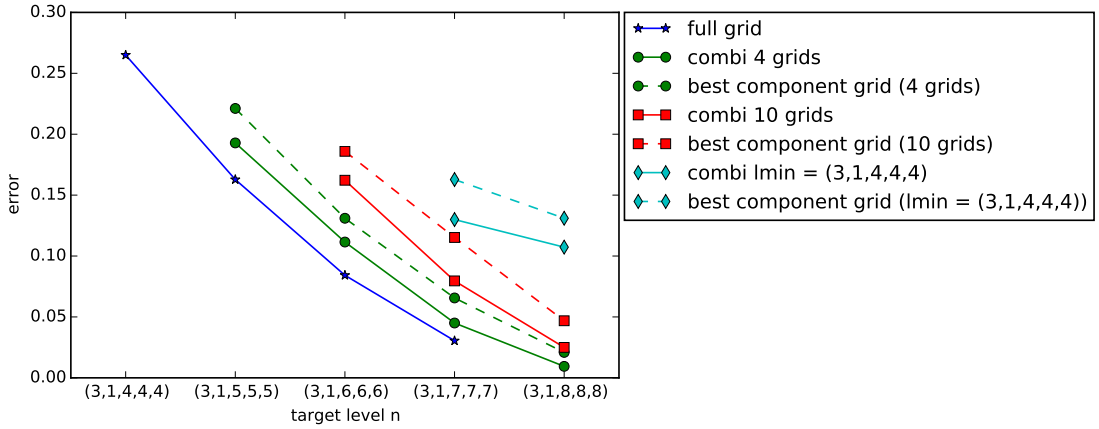

Figure 5.14.: Error of g compared to the full grid solution with $\vec{l} = (3, 1, 8, 8, 8)$. The combined solutions were produced by combining once at end of the simulation and the phase and amplitude of the component solutions was normalized.

Figure 5.14 shows the error of the combined solutions for the case of combining once in the end of the simulation and normalizing phase and amplitude. Although the error is higher than when combining after each time step, the combined solutions converge and are better than the corresponding best component grid. Thus, with the here proposed normalizations it is possible to successfully apply the combination technique even when only combining once in the end.

Accuracy in relation to computational cost

Figures 5.15 and 5.16 (left) show the relative increase of the error for different lengths of the combination interval compared to the best case, i.e., combining after each time step. This means, the errors for g obtained with different combination intervals for a particular combination technique are divided by the error that was obtained with this particular combination technique with a combination interval of one. For the combination once in the end (6000 time steps), the error for the case where the phase and amplitude were normalized was used. For the other combination intervals, no normalization was

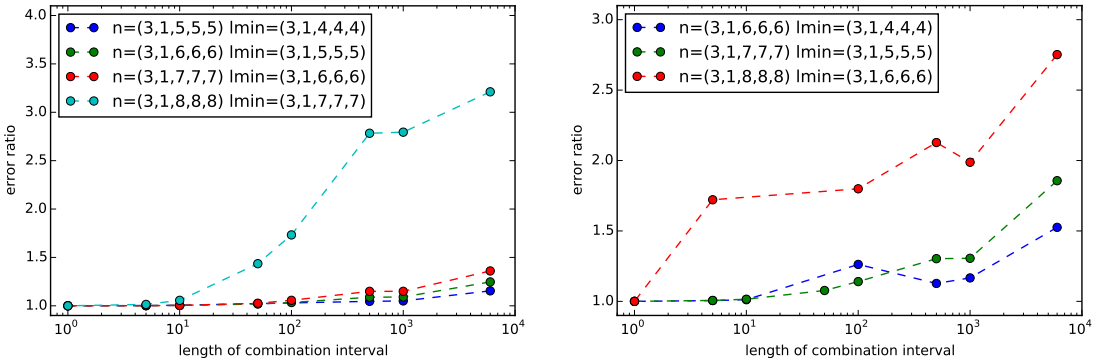

Figure 5.15.: Relative increase of the error for g for different combination intervals compared to the case when combining after each time step.

used. Furthermore, the combination intervals that were obviously problematic (compare Figures 5.12 and 5.13) were excluded here. One can observe that the error ratio increases not only with the length of the combination interval, but also with the target level.

Figure 5.16 (right) shows the runtime for 6000 time steps with the combination technique $\vec{n} = (3, 1, 8, 8, 8)$ and $\vec{l}_{\min} = (3, 1, 4, 4, 4)$ for different lengths of the combination interval. Furthermore, it shows the aggregate times for the actual computations in GENE (*run GENE*; this is the *solve* step in Algorithm 3), the conversion of the simulation data to the distributed component grid data structure and back (*convert*), and the time for the combination step (*combine*). For comparison, also the time to compute 6000 time steps on the full grid (*fg*) is included. The experiments with were performed with four process groups which had 16 processes each. The full grid was computed with 64 processes and so with the same total number of processes as the combination technique experiments.

One can observe that the time for the actual computations increases significantly when the combination interval becomes smaller. This effect is caused by overhead in GENE that occurs each time when a component grid is restarted after the combination step. Each time GENE is restarted, it creates all the required data structures and runs different initialization routines. For this very simple simulation scenario, the compute time per time step is particularly low and the overhead is in the order of multiple time steps. Thus, the overhead dominates the time of the computations for a combination interval of 10 time steps or less. For more complex simulation scenarios it can be expected that the time per time step is significantly higher and thus this effect is less severe. The overhead in GENE is a result of the black-box approach, where only the absolutely necessary modifications were made in the GENE code. A way to reduce this overhead would be to modify the GENE code so that the most expensive data structures are externally stored in the interface class, so that they can be reused and do not have to be recreated after each combination step.

The time for the combination step and for the conversion operations increases linearly with the number of combination steps. For this simple application scenario the time per time step for the computations is particularly small in comparison to the time for

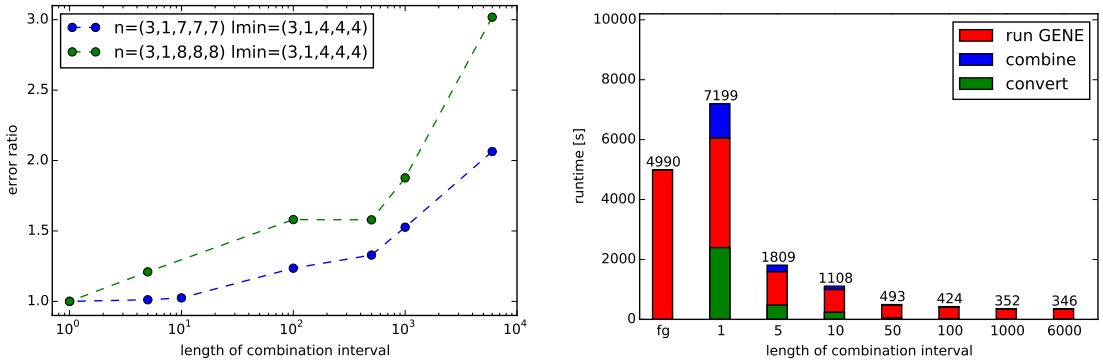

Figure 5.16.: **Left:** Relative increase of the error for g for different combination intervals compared to the case when combining after each time step. **Right:** Computation times for different combination intervals with $\vec{n} = (3, 1, 8, 8, 8)$ and $\vec{l}_{\min} = (3, 1, 4, 4, 4)$ and for the full grid reference solution with $\vec{n} = (3, 1, 8, 8, 8)$.

these operations and thus these steps significantly contribute to the total time when combining after each time step. Furthermore, this first implementation of the conversion operations is not very efficient. For a more complex simulation scenario, with higher costs per time step, and a for higher target resolutions with more and larger component grids, the contribution of the combination step (and the conversions) is expected to be much less significant (compare the estimates for a large GENE problem in Section 3.3).

In general, for the selection of the combination interval, there is a trade off between computational resources and error reduction. For the here presented experiments, in most cases the highest absolute error reduction is already achieved by reducing the combination interval of 6000 time steps to 1000 time steps. In the case $\vec{n} = (3, 1, 8, 8, 8)$, $\vec{l}_{\min} = (3, 1, 4, 4, 4)$, the error reduces by 37% from 0.107 (6000 time steps and normalization) to 0.067 (1000 time steps). However, the total compute time increases only by less than 2%. Using a combination interval of 100 time steps reduces the error by 47% from 0.107 to 0.0562, but increases the computational cost already by 23%. Further reducing the error by using a combination interval of 5 or 1 time steps becomes so expensive that for this simulation scenario it might be more economic to increase the target resolution instead. As it can be seen in Figure 5.12, using higher target levels was more beneficial for the accuracy than using smaller combination intervals.

The fact that it is possible to compute the full grid solution with such a low number of processes indicates that the computational demand of this application scenario is not in a range where one would expect a huge benefit of the combination technique. However, even with this small problem, and only a low number of dimensions used for the combination technique, for a combination interval of 100 time steps or higher, the difference in runtime between the full grid solution and the combination technique was already more than a factor of ten. It is expected that this difference will even increase for larger and more complex simulation scenarios and more dimensions being used for

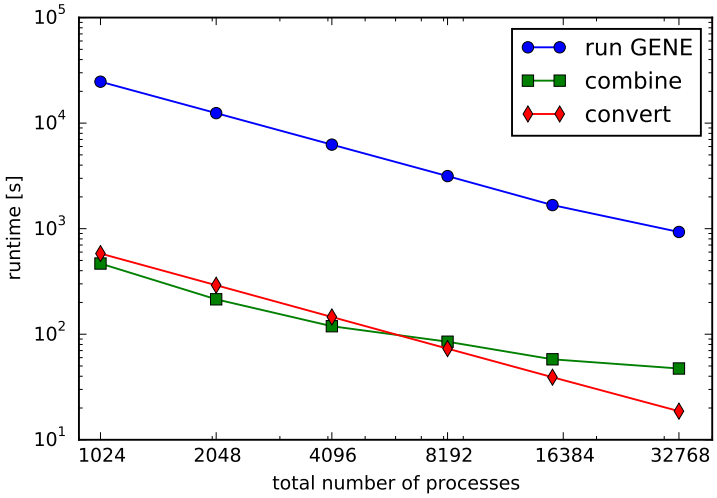

Figure 5.17.: Runtime for different numbers of processes for a combination technique with $\vec{n} = (9, 1, 13, 13, 13)$ and $\vec{l}_{\min} = (9, 1, 4, 4, 4)$. The x-axis of the plot shows the total number of processes, i.e., the number of process groups multiplied with the process group size.

the combination technique. For large-scale gyrokinetic simulations, which are restricted by limited computational resources either in resolution or in the number of time steps that can be computed, a factor of ten is already a significant improvement that can make the difference whether the computation of a particular simulation scenario is feasible or not. With the above experiments a major bottleneck was identified: In order to enable efficient computations with short combination intervals, the way how GENE interacts with the combination technique framework has to be improved.

Large-scale experiments

Figure 5.17 shows the runtimes for a large-scale experiment with $\vec{n} = (9, 1, 13, 13, 13)$ and $\vec{l}_{\min} = (9, 1, 4, 4, 4)$. The combination scheme consists of 136 component grids. The computations were done for 300 time steps in total with a combination interval of 100 time steps. Here the resolution of the k_x-direction was increased in order to virtually increase the problem size so that it is possible to use a large number of processes efficiently. The experiments were performed with a process group size of 1024 processes. The number of process groups was then varied to study the strong scaling behavior.

One can observe that the time for the computations in GENE (*run GENE*) and for the conversion step (*convert*) scales almost linearly. This is related to the coarse level of parallelism of the combination technique which ensures scalability for these local operations (local in the sense that the operation does only involve communication within one process group) as long as there are enough component grids per process group so that a good load balance can be obtained (compare Section 3.2). The runtime for

the combination step (*combine*) shows the same behavior that was already observed in previous experiments. For a discussion of the scaling behavior of the combination step refer to Section 3.3.

The purpose of this experiment was to demonstrate the potential of the combination technique to achieve target resolutions with GENE that are out of reach for the direct computation on the full grid. Directly computing a grid with $512 \times 1 \times 8192 \times 8192 \times 8192$ grid points would require an aggregate main memory size of more than 4.5 petabyte, just to store the computational grid. Note that this is more than four times the aggregate size of main memory that is available on *Hazel Hen* (987 TB, compare Appendix A). With the combination technique it is possible to compute such an extremely large problem with only 1024 processes. The size of the component grids ranges between 8.5 GB and 32 GB. This results in an aggregate size of roughly 3 TB of main memory to store the component grids.

Note that the actual demand for main memory might be significantly higher than the amount of memory that is required to store the computational grids. In order to perform the computations, GENE creates various temporary data structures that additionally consume a significant amount of main memory. Thus, the size of the computational grids gives only a lower bound for the actual demand of main memory. This holds for both the combination technique and the direct computation on the full grid.

Visualization of the combined solution

Figure 5.18 shows an exemplary reference solution and combined solution with the same target level. The plots show a two-dimensional slice at $\mu = 0.25$ (arbitrarily chosen) of the absolute value of g at the end of the simulation. In $v_\parallel$-direction ('v' in the plot), the numbers indicate the indices of the grid points. The z-k_x-axis corresponds to the z-direction for the k_x-modes with indices 0 and 1. This, means the first half of the plot (the first 32 points on the z-k_x-axis) corresponds to the z-$v_\parallel$-plane for the k_x-mode with index 0 and the second half of the plot corresponds to the k_x-mode with index 1. Both halves are connected through the boundary condition in z-direction (see Section 5.1.2). As out of the eight wave numbers in k_x-direction only those with the indices -1, 0 and 1 contain values different than zero, for higher modes the solution field is zero.

The combined solution was obtained by combination after each time step. The differences between the reference solution and the combined solution are hardly visible. Both solutions were normalized according to Equation (5.22), so that they have the same amplitude.

Figure 5.18 also shows the component solutions corresponding to the combination scheme. However, note that these solutions were obtained by individually computing the component grids until convergence, without intermediate combination steps, and thus do not correspond to the combined solution depicted here, which was computed with a combination interval of one time step. They correspond to the case where the component solutions are combined once in the end. These component solutions are included here for a visual demonstration that the component grids are not just simple projections of the reference solution onto coarser grids. Although all these component solutions are

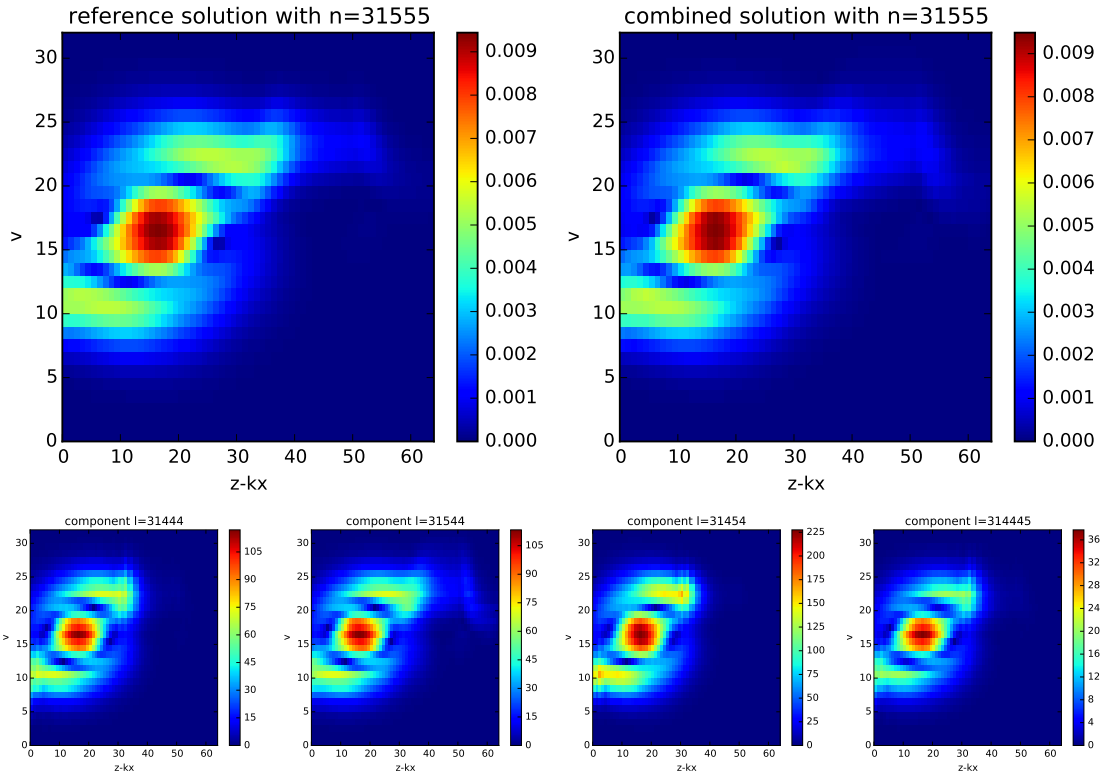

Figure 5.18.: **Top:** Reference solution and combined solution with $\vec{n} = (3, 1, 5, 5, 5)$ and $\vec{l}_{\min} = (3, 1, 4, 4, 4)$. **Bottom:** Corresponding component solutions.

different from each other and from the reference solution, with the combination technique it is possible to obtain an approximation which is closer to the reference solution than any of the component solutions. The component solutions were computed for the same number of time steps and time step size. They were not normalized so that one can easily see the significantly different amplitudes. This visualizes why normalization is necessary when the component solutions are only combined once in the end.

5.2.3. Other quantities of interest in GENE

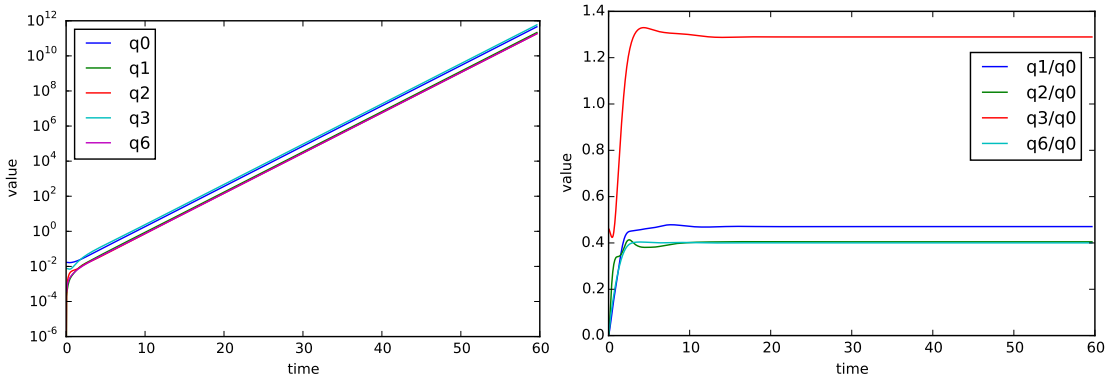

Figure 5.19.: Quantities $\tilde{q}_i$ over time for the reference full grid solution with $\vec{l} = (3, 1, 8, 8, 8)$. On the right hand side the quantities are normalized by $\tilde{q}_0$.

Other important quantities computed in GENE are the velocity space moments of the particle distribution function and the particle, heat and momentum fluxes, that are output in the file `nrg.dat`. The combination technique was applied to selected quantities $\tilde{q}_i$ in this file

$$\tilde{q}_0 := \frac{\langle |n_1|^2 \rangle}{(n_0 \rho^*_{\text{ref}})^2}, \qquad \tilde{q}_1 := \frac{\langle |u_{1\parallel}|^2 \rangle}{(v_T \rho^*_{\text{ref}})^2}, \qquad \tilde{q}_2 := \frac{\langle |T_{1\parallel}|^2 \rangle}{(T_0 \rho^*_{\text{ref}})^2},$$
$$\tilde{q}_3 := \frac{\langle |T_{1\perp}|^2 \rangle}{(T_0 \rho^*_{\text{ref}})^2}, \qquad \tilde{q}_6 := \frac{\langle Q^x_{\text{es}} \rangle}{Q_{\text{gb}}},$$

where i corresponds to the column number in the file. These quantities relate to the three-dimensional values

$$n_1(x,y,z) = \int g \, \mathrm{d}^3 v, \qquad T_{1\perp}(x,y,z) = \frac{m}{2n_0} \int v_\perp^2 g \, \mathrm{d}^3 v - T_0 \frac{n_1}{n_0},$$
$$u_{1\parallel}(x,y,z) = \frac{1}{n_0} \int v_\parallel g \, \mathrm{d}^3 v, \qquad Q(x,y,z) = \int \frac{1}{2} m v^2 g v_D \, \mathrm{d}^3 v,$$
$$T_{1\parallel}(x,y,z) = \frac{m}{n_0} \int (v_\parallel - u_{1\parallel})^2 g \, \mathrm{d}^3 v - T_0 \frac{n_1}{n_0},$$

where the integration in the velocity directions is expressed only symbolically. Due to the gyrokinetic formulation, the actual representation of these formulas would be more involved. The brackets $\langle \cdot \rangle$ indicate the average over the remaining three spatial dimensions. The above formulations follow the documentation of GENE. For a detailed derivation of these values please refer to [34]. As this application scenario uses adiabatic electrons, the other quantities in the file were not relevant and thus were not considered here.

All these values grow exponentially over time with the same rate as g. Thus they can be used in order to normalize the amplitude of the component solutions (compare Section 5.2.2). In the following, a normalization with the value $\tilde{q}_0$ is used to make the other quantities time-independent. Thus, as soon as the simulation is converged, they will not change their value any more. This is visualized in Figure 5.19.

The combined solution is then computed as

$$q_i^{(c)} = \sum_{\vec{l} \in \mathcal{I}} c_{\vec{l}} \frac{\tilde{q}_{\vec{l},i}}{\tilde{q}_{\vec{l},0}}. \tag{5.25}$$

The error to a given reference solution is computed as

$$e_{q_i} = |q_i - q_{\mathrm{ref},i}|. \tag{5.26}$$

Figures 5.20 to 5.23 show the error of the different combined solutions. The values were outputs of the experiments that were performed to compute the eigenvalue λ, described in Section 5.2.1. The results for the combination of these quantities are similar to the results for the eigenvalues and eigenvectors: one can observe that the combined solutions converge and that (except for one case) they are better than the best component grid. The only outliers can be seen for q_2 in Figure 5.21 (left), where in one case the combined solution was better than the corresponding full grid solution, and in Figure 5.21 (right), where in one case the best component grid was better than the combined solution. The results indicate that the combination technique can successfully be applied also to these quantities of interest in GENE.

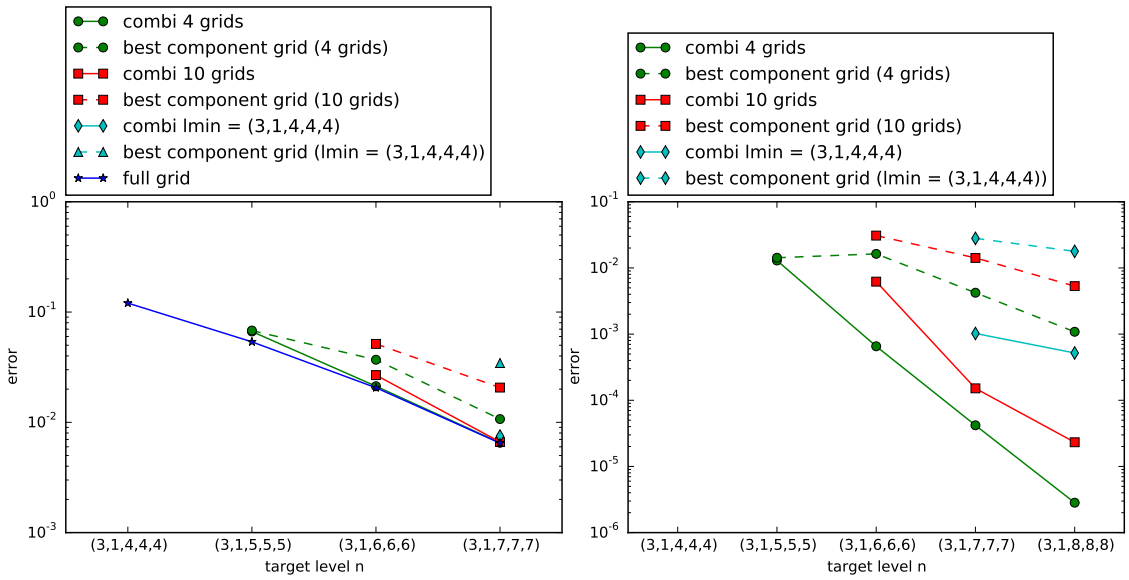

Figure 5.20.: **Left:** Error of q_1 compared to the full grid solution with $\vec{n} = (3, 1, 8, 8, 8)$. **Right:** Error of q_1 compared to the full grid solution with the corresponding target level $\vec{n}$.

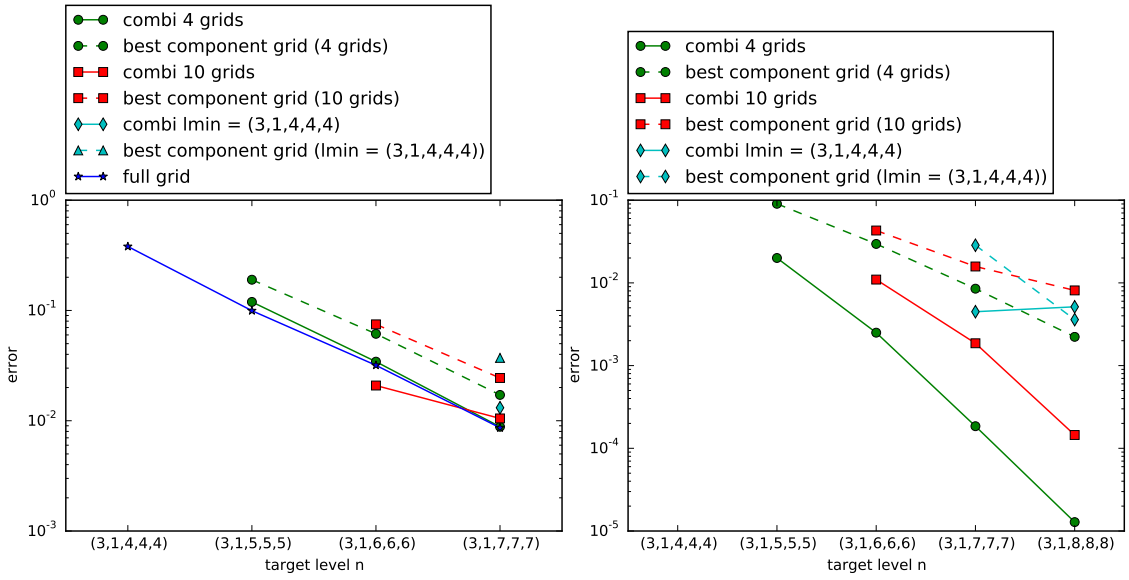

Figure 5.21.: **Left:** Error of q_2 compared to the full grid solution with $\vec{n} = (3, 1, 8, 8, 8)$. **Right:** Error of q_2 compared to the full grid solution with the corresponding target level $\vec{n}$.

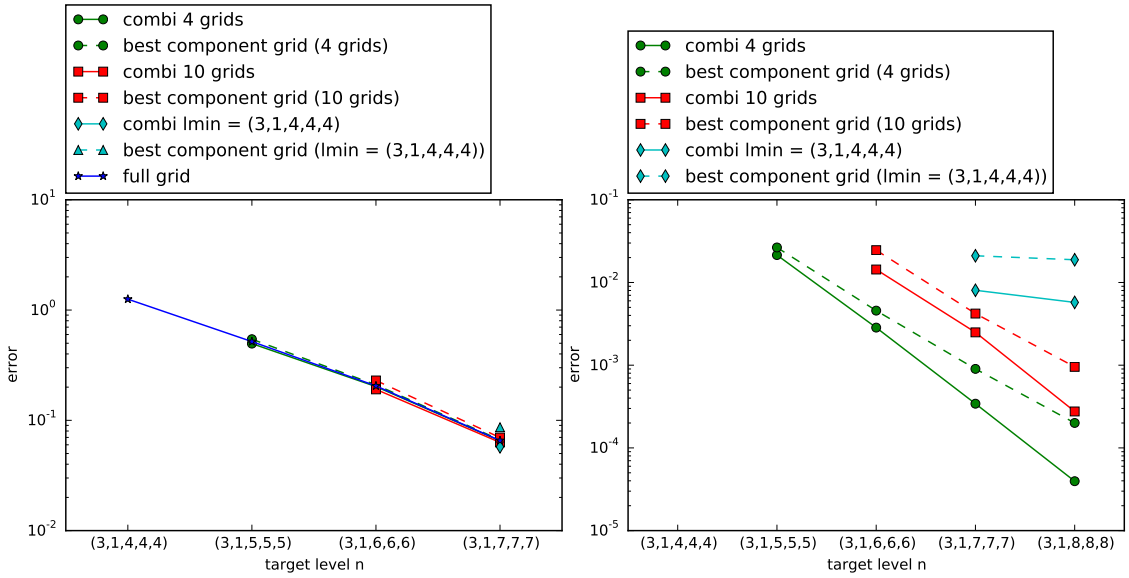

Figure 5.22.: **Left:** Error of q_3 compared to the full grid solution with $\vec{n} = (3, 1, 8, 8, 8)$. **Right:** Error of q_3 compared to the full grid solution with the corresponding target level $\vec{n}$.

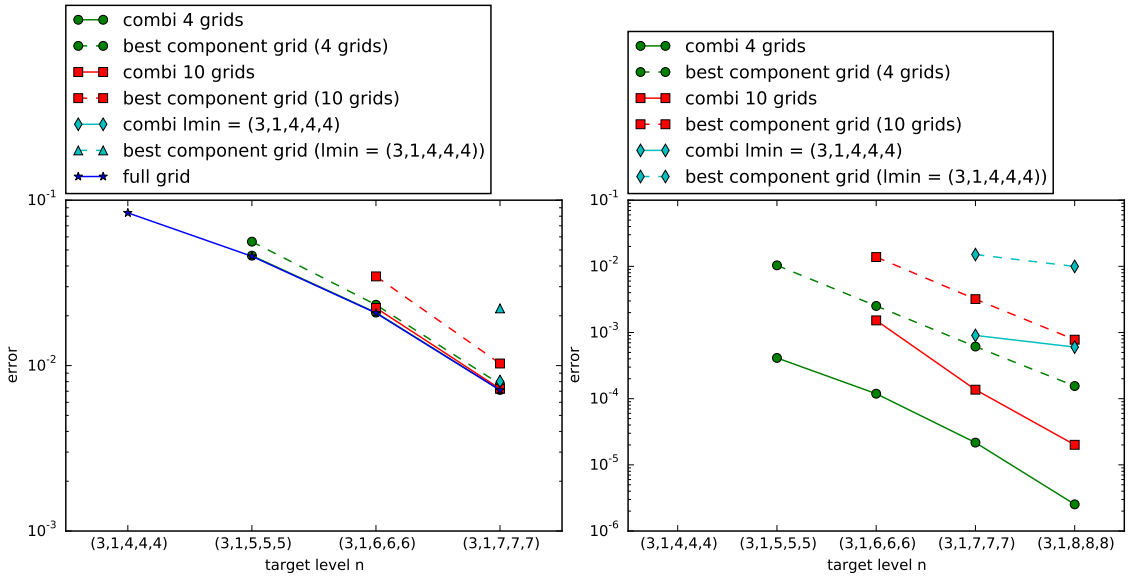

Figure 5.23.: **Left:** Error of q_6 compared to the full grid solution with $\vec{n} = (3, 1, 8, 8, 8)$. **Right:** Error of q_6 compared to the full grid solution with the corresponding target level $\vec{n}$.

5.3. Outlook

The experiments demonstrate that the combination technique can be successfully applied to local linear initial-value simulations in GENE. However, the computational demand of the simulation scenario used for the experiments is too low to enable a meaningful investigation of the performance benefit of the combination technique.

Global simulations would be much better suited for this purpose, as they require a very high number (several thousand) of grid points in x-direction (in the global case no spectral representation is used in this direction). Thus, not only the target resolution is significantly higher than in the local case, but also there is a real benefit of considering the x-direction for the combination technique. However, there are impediments of technical nature that have to be solved before these kinds of simulations are possible with the combination technique. As already discussed in the previous section, GENE has to be modified in order to enable an efficient restart after the combination step. In the case of global simulations, a critical issue is the so-called *gyroaveraging matrix*, which is created each time GENE is started (or restarted for the next outer iteration). The initialization of this matrix is not very well optimized and consumes a substantial amount of time (in the order of minutes or even hours for large problem sizes) and main memory. This is usually not a problem for the typical use cases in GENE where several hundred thousand time steps are computed and the whole simulation run takes many hours or even days. However, the initialization cost of the gyroaveraging matrix makes experiments with the combination technique, where only a few time steps are simulated during each outer iteration, infeasible. Therefore, as a next step towards global initial-value simulations with the combination technique, it is important to avoid that the gyroaveraging matrix is newly created each time. This can be achieved in a similar manner as it is done with the checkpoints: after the matrix has been created for a particular component grid it is stored in the main memory of the process group and then copied back whenever this component grid is restarted for the next outer iteration.

The eventual goal of EXAHD is to be able to apply the combination technique to global non-linear initial-value simulations in GENE. Here, also a considerable number of grid points is used in the k_y dimension, which enables a fully five-dimensional combination technique and thus the potential gain in target resolution with the combination technique compared to direct computations on the full grid is enormous. Although there is no experience with the combination technique applied to non-linear initial-value simulations in GENE so far, the development of the algorithms and data structures presented in this work has laid important groundwork to enable these kinds of experiments in the near future.

6. Conclusions

In this work a combination technique was presented that is capable of the massively parallel and fault-tolerant solution of time-dependent high-dimensional PDEs on large-scale HPC systems. Experiments with the software GENE demonstrate that the here presented combination technique has the potential to significantly push the computational limits of gyrokinetic simulations.

In this work the most important aspects for the design of a parallel combination technique were addressed. Investigating how to combine the component grids most efficiently, it was found that the best way to do this is to use the hierarchical basis. The result of these investigations were new communication strategies that significantly reduce the communication effort of the combination step.

Another crucial element for massively parallel computations with the combination technique is load balancing. Experiments with GENE have shown that the anisotropy of the discretization can result in severely different runtimes, even for component grids that have the same number of grid points. On this basis, the manager-worker scheme was identified to be the best suitable parallelization concept for the combination technique in terms of load balancing. Furthermore, a general cost model was proposed that is able to improve the load balance over a standard model.

The major contribution of this work are the efficient and scalable algorithms and data structures for the combination step. Finding an efficient algorithm for the hierarchization and dehierarchization of the component grids that are stored in a distributed memory fashion was crucial to enable the combination in the hierarchical basis. Furthermore, suitable data structures for a distributed sparse grid were presented that enable the local and global reduction of the combined solution in a scalable way.

Fault-tolerance is expected to become an important factor for the design of numerical algorithms for future exascale system. In this work a parallel implementation of the fault-tolerant combination technique was presented and it was demonstrated that this approach enables algorithm-based fault-tolerance on massively parallel systems. In order to be able to perform experiments with the fault-tolerant combination technique on HPC systems where no fault-tolerant MPI implementation is available, a fault simulation software layer was developed. The fault simulation layer is a general library and is independent of a specific application. Thus, it can also be used to perform experiments with other fault-tolerant algorithms.

The parallel combination technique was applied to gyrokinetic simulations with GENE. In experiments with local linear initial-value simulations, it was found that the combination technique can be successfully applied to the eigenvalue, the eigenvector and other quantities of interest that are the outcomes of these simulations. Furthermore, these experiments have shown that for the eigenvector the accuracy of the combined solution

increases with the frequency of the combination step. In a large-scale experiment, the potential of the combination technique to significantly increase the target resolution of GENE simulations beyond what is feasible with the direct computation on the full grid was demonstrated. The software framework for the parallel combination technique developed in this work serves as a basis to enable future large-scale experiments with more sophisticated application scenarios in GENE, such as global non-linear initial-value simulations of a large fusion device such as ITER, that have so far been out of reach due to their prohibitive computational demand.

A. HPC systems used for the experiments

Hazel Hen

Hazel Hen was a Cray XC40 system located at the High Performance Computing Center Stuttgart (HLRS). It consists of 7712 dual-socket nodes which have 128 GB of main memory each. Each socket is equipped with a Intel Xeon E5-2680 v3 (Haswell) CPU which has 12 cores and 30MB L3 cache. This results in a total number of 185,088 compute cores and a combined main memory size of 987 TB. The nodes are connected with Cray's *dragonfly* network topology, which is neither a classical torus network nor a tree topology. The network is built of the Aries interconnect which have three different types of links. Each interconnect is connected to 4 computed nodes. With the first type of link all the interconnects within a chassis are connected to each other. Six chassis form a group. With the second type of link, each interconnect is connected to the all the other interconnects that have the same relative position in the other chassis of the group. The third type of link is used to connect the interconnects of different groups.

Hermit

Hermit is a Cray XE6 system that was located at the High Performance Computing Center Stuttgart (HLRS). By now, this system has been replaced by *Hazel Hen*. 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 each.

Hornet

Hornet was the direct predecessor of *Hazel Hen*. It consisted of identical Hardware, but had only 3944 nodes.

SuperMUC Phase 1

SuperMUC Phase 1 is located at the Leibnitz-Rechenzentrum (LRZ) and consists of 9126 thin nodes suited for massively parallel applications and some special purpose nodes. 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.

B. GENE parameter files used in the experiments

B.1. Linear local simulation

```
&parallelization
n_procs_s =   1
n_procs_v =   1
n_procs_w =   256
n_procs_x =   1
n_procs_y =   1
n_procs_z =   2
/

&box
n_spec =    1
nx0    =   8
nky0   =   1
nz0    =   256
nv0    =   256
nw0    =   256

kymin =   0.3000
lv    =     3.00
lw    =     9.00
adapt_lx = T
ky0_ind =   1
mu_grid_type = 'equidist'
/

&in_out
diagdir = './'
chptdir = './'

read_checkpoint  = F
write_checkpoint = T
istep_field =     1
istep_mom   =     -100
istep_nrg   =      10
istep_omega =     20
istep_vsp   =     -500
istep_schpt =    -500
istep_energy =    -500

write_std = T
write_h5 = F
chpt_h5 = F
momentum_flux = F
/

&general
nonlinear =   F
comp_type = 'IV'
perf_vec  =  2 2 1 1 1 1 2 1 2
!nblocks   =      16
arakawa_zv = T
arakawa_zv_order = 2
hypz_opt = F

timescheme = 'RK4'
dt_max     = 0.01

timelim = 10000
ntimesteps = 10000
calc_dt = T
omega_prec = 1e-6
underflow_limit = 1e-12
```

```
beta       =    0.0000000
debye2     =    0.0000000
collision_op = 'none'

init_cond = 'alm'

hyp_z =   -1
hyp_v =   0.2000

perf_tsteps =  20

/

&geometry
magn_geometry = 's_alpha'
q0       =    1.4000000
shat     =   0.7960
trpeps   =   0.18000000
major_R  =    1.0000000
norm_flux_projection  = F
/

&species
name   = 'ions'
omn    =    2.2200000
omt    =    6.9200000

mass   =    1.0000000
temp   =    1.0000000
dens   =    1.0000000
charge =  1
/
```

B.2. Non-linear global simulation

```
&parallelization
n_procs_s = 2
n_procs_v = 1
n_procs_w  = 16
n_procs_x = 16
n_procs_y = 1
n_procs_z = 16
n_parallel_sims = 1
/

&box
n_spec =     2
nx0    =   1024
nky0   =   32
nz0    =   32
nv0    =   64
nw0    =   64

n0_global = 20

kymin = 0.05
lv     =      5.0
lw     =     16.0
lx     =    240.0
x0     =      0.5
mu_grid_type = 'equidist'
/

&in_out
diagdir = './out'

read_checkpoint  = F
write_checkpoint = F

istep_field =     100
istep_mom   =     1000
istep_nrg   =      10
istep_vsp   =       0
istep_schpt =   20000
/

&general

nonlinear = T
```

```
x_local   = F

calc_dt = T

dt_max     =   0.0125
courant    =     0.30

timelim    =      32000
ntimesteps = 10
underflow_limit =   0.1000E-09

beta       =    0.001
debye2     =    0.000

collision_op = 'none'

init_cond = 'db'

hyp_x =    0.000
hyp_z =    1.000
hyp_v =    0.200

!perf_vec  =  2 2 1 1 1 1 2 1 2
/

&nonlocal_x
/

&geometry
magn_geometry = 'circular'
!q0      =   1.500
!trpeps  =   0.1800
minor_r  =   0.3600
major_R  =   1.000
mag_prof =   T
q_coeffs =   0.9500, 0.0,  2.200
rhostar  =   0.004 !1/250
/

&species
name   = 'ions'
prof_type =  3
kappa_T   =  6.9000
LT_center =  0.5000
LT_width  =  0.5000E-01

kappa_n   =  3.0
Ln_center =  0.5000
Ln_width  =  0.5000E-01

delta_x_T  = 0.2500
delta_x_n  = 0.2500

mass   =    1.000
temp   =    1.000
dens   =    1.000
charge =  1
/

&species
name   = 'electrons'
prof_type =  3
kappa_T   =   6.900
LT_center =   0.5000
LT_width  =   0.5000E-01

kappa_n   =   3.00
Ln_center =   0.5000
Ln_width  =   0.5000E-01

delta_x_T  =  0.2500
delta_x_n  =  0.2500

mass   =    5.446E-4
temp   =    3.
dens   =    1.000
charge = -1
/
```

Bibliography

[1] The GENE code. http://www.genecode.org/.

[2] German priority programme 1648: Software for exascale computing. http://www.sppexa.de.

[3] The ITER project. http://www.iter.org.

[4] SG++ software library. http://sgpp.sparsegrids.org.

[5] Unified european applications benchmark suite. http://www.prace-ri.eu/ueabs/.

[6] M. Ali. *High Performance Fault-Tolerant Solution of PDEs using the Sparse Grid Combination Technique*. PhD thesis, Australian National University, 2016.

[7] M. M. Ali, J. Southern, P. Strazdins, and B. Harding. Application level fault recovery: Using fault-tolerant Open MPI in a PDE solver. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1169–1178, May 2014.

[8] Md Mohsin Ali, Peter Strazdins, Brendan Harding, and Markus Hegland. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *The International Journal of High Performance Computing Applications*, 30(3):335–359, 2016.

[9] M.M. Ali, P.E. Strazdins, B. Harding, M. Hegland, and J.W. Larson. A fault-tolerant gyrokinetic plasma application using the sparse grid combination technique. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 499 – 507. IEEE, 2015.

[10] Akio Arakawa. Computational design for long-term numerical integration of the equations of fluid motion: two-dimensional incompressible flow. part i. *Journal of Computational Physics*, 135(2):103–114, 1997.

[11] B.R. Baliga and S.V. Patankar. A new finite-element formulation for convection-diffusion problems. *Numerical Heat Transfer*, 3(4):393–409, 1980.

[12] Janos Benk and Dirk Pflüger. Hybrid parallel solutions of the black-scholes PDE with the truncated combination technique. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 678–683. IEEE, 2012.

[13] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for user-level failure mitigation in the MPI-3 standard.

Department of Electrical Engineering and Computer Science, University of Tennessee, 2012.

[14] Markus Blatt, Ansgar Burchardt, Andreas Dedner, Christian Engwer, Jorrit Fahlke, Bernd Flemisch, Christoph Gersbacher, Carsten Gräser, Felix Gruber, Christoph Grüninger, et al. The distributed and unified numerics environment, version 2.4. *Archive of Numerical Software*, 4(100):13–29, 2016.

[15] A.J. Brizard and T.S. Hahm. Foundations of nonlinear gyrokinetic theory. *Rev. Modern Physics*, 79(421):421–468, 2007.

[16] Hans-Joachim Bungartz and Michael Griebel. Sparse Grids. *Acta Numerica*, 13:147–269, 2004.

[17] Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel, editors. *Software for Exascale Computing - SPPEXA 2013-2015*. Lecture Notes in Computational Science and Engineering. Springer, 2016.

[18] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[19] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[20] Zizhong Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. *SIGPLAN Not.*, 48(8):167–176, February 2013.

[21] Edward Coffman, Michael Garey, and David Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[22] T. Dannert. *Gyrokinetische Simulation von Plasmaturbulenz mit gefangenen Teilchen und elektromagnetischen Effekten*. PhD thesis, Technische Universität München, 2004.

[23] John D'Errico. Matlab toolbox: polyfitn, 2012. https://de.mathworks.com/matlabcentral/fileexchange/34765-polyfitn.

[24] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. The international exascale software project roadmap. *The international journal of high performance computing applications*, 25(1):3–60, 2011.

[25] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and T Brightwell. rMPI: increasing fault resiliency in a message-passing environment. *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011.

[26] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[27] X. Garbet, Y. Idomura, L. Villard, and T. Watanabe. Gyrokinetic simulations of turbulent transport. *Nuclear Fusion*, 2010.

[28] Jochen Garcke. Regression with the optimised combination technique. In *Proceedings of the 23rd international conference on Machine learning*, pages 321–328. ACM, 2006.

[29] Jochen Garcke. A dimension adaptive sparse grid combination technique for machine learning. In Wayne Read, Jay W. Larson, and A. J. Roberts, editors, *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006*, volume 48 of *ANZIAM J.*, pages C725–C740, December 2007.

[30] Jochen Garcke. An optimised sparse grid combination technique for eigenproblems. *PAMM*, 7(1):1022301–1022302, 2007.

[31] Thomas Gerstner and Michael Griebel. Numerical integration using sparse grids. *Numerical Algorithms*, 18(3-4):209–232, 1998.

[32] Thomas Gerstner and Michael Griebel. Dimension–adaptive tensor–product quadrature. *Computing*, 71(1):65–87, 2003.

[33] T Görler, X Lapillonne, Stephan Brunner, Tilman Dannert, Frank Jenko, Florian Merz, and D Told. The global version of the gyrokinetic turbulence code GENE. *Journal of Computational Physics*, 230(18):7053–7071, 2011.

[34] Tobias Görler. *Multiscale effects in plasma microturbulence*. PhD thesis, Universität Ulm, 2009.

[35] V. Gradinaru. Fourier transform on sparse grids: Code design and the time dependent schrödinger equation. *Computing*, 80(1):1–22, May 2007.

[36] M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl. The combination technique for parallel sparse-grid-preconditioning or -solution of PDEs on workstation networks. In *Parallel Processing: CONPAR 92 VAPP V*, volume 634 of *LNCS*, 1992.

[37] Michael Griebel and Jan Hamaekers. *Fast Discrete Fourier Transform on Generalized Sparse Grids*, pages 75–107. Springer International Publishing, Cham, 2014.

[38] Michael Griebel, Walter Huber, and C Zenger. Numerical turbulence simulation on a parallel computer using the combination method. In *Flow Simulation with High-Performance Computers II*, pages 34–47. Springer, 1996.

[39] Michael Griebel, Michael Schneider, and Christoph Zenger. A combination technique for the solution of sparse grid problems. In Pieter de Groen and Robert Beauwens,

editors, *Proceedings of the IMACS International Symposium on Iterative Methods in Linear Algebra: Brussels, Belgium, 2 - 4 April, 1991*. IMACS, North Holland, 1992.

[40] Michael Griebel and Veronika Thurner. The efficient solution of fluid dynamics problems by the combination technique. *International Journal of Numerical Methods for Heat & Fluid Flow*, 5(3):251–269, 1995.

[41] Klaus Hallatschek. Fouriertransformation auf dünnen Gittern mit hierarchischen Basen. *Numerische Mathematik*, 63(1):83–97, 1992.

[42] B. Harding, M. Hegland, J. Larson, and J. Southern. Scalable and Fault Tolerant Computation with the Sparse Grid Combination Technique. *ArXiv e-prints*, April 2014.

[43] Brendan Harding. *Fault Tolerant Computation of Hyperbolic Partial Differential Equations with the Sparse Grid Combination Technique*. PhD thesis, Mathematical Sciences Institute, The Australian National University, 2016.

[44] Brendan Harding and Markus Hegland. A parallel fault tolerant combination technique. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 584–592, 2013.

[45] Brendan Harding and Markus Hegland. A robust combination technique. *ANZIAM Journal*, 54:C394–C411, 2013.

[46] Brendan Harding and Markus Hegland. Robust solutions to PDEs with multiple grids. In *Sparse Grids and Applications-Munich 2012*, pages 171–193. Springer, 2014.

[47] Brendan Harding, Markus Hegland, Jay Larson, and James Southern. Fault tolerant computation with the sparse grid combination technique. *SIAM Journal on Scientific Computing*, 37(3):C331–C353, 2015.

[48] Mario Heene, Alfredo Parra Hinojosa, Hans-Joachim Bungartz, and Dirk Pflüger. A massively-parallel, fault-tolerant solver for high-dimensional PDEs. In *Euro-Par 2016: Parallel Processing Workshops*, volume 10104 of *Lecture Notes in Computer Science*, pages 635–647. Springer, 2016.

[49] Mario Heene, Christoph Kowitz, and Dirk Pflüger. Load balancing for massively parallel computations with the sparse grid combination technique. In Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 574–583. IOS Press, 2014.

[50] Mario Heene and Dirk Pflüger. Efficient and scalable distributed-memory hierarchization algorithms for the sparse grid combination technique. In *Parallel Computing: On the Road to Exascale*, volume 27 of *Advances in Parallel Computing*, pages 339–348. IOS Press, 2015.

[51] Mario Heene and Dirk Pflüger. Scalable algorithms for the solution of higher-dimensional PDEs. In Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel, editors, *Software for Exascale Computing - SPPEXA 2013-2015*, volume 113 of *Lecture Notes in Computational Science and Engineering*, pages 165–186. Springer, 2016.

[52] M. Hegland. Adaptive sparse grids. In K. Burrage and Roger B. Sidje, editors, *Proc. of 10th Computational Techniques and Applications Conference CTAC-2001*, volume 44, pages C335–C353, April 2003.

[53] Markus Hegland, Jochen Garcke, and Vivien Challis. The combination technique and some generalisations. *Linear Algebra and its Applications*, 420(2):249 – 275, 2007.

[54] Alfredo Parra Hinojosa, Brendan Harding, Markus Hegland, and Hans-Joachim Bungartz. Handling silent data corruption with the sparse grid combination technique. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 187–208. Springer, 2016.

[55] Alfredo Parra Hinojosa, Christoph Kowitz, Mario Heene, Dirk Pflüger, and Hans-Joachim Bungartz. Towards a fault-tolerant, scalable implementation of GENE. In *Recent Trends in Computational Engineering – CE2014*, volume 105 of *Lecture Notes in Computational Science and Engineering*, pages 47–65. Springer, 2015.

[56] P. Hupp and R. Jacob. A cache-optimal alternative to the unidirectional hierarchization algorithm. *Proceedings of the Workshop Sparse Grids and Algorithms 2014*, 2014.

[57] Philipp Hupp. Performance of unidirectional hierarchization for component grids virtually maximized. *Procedia Computer Science*, 29:2272–2283, 2014.

[58] Philipp Hupp, Mario Heene, Riko Jacob, and Dirk Pflüger. Global communication schemes for the numerical solution of high-dimensional PDEs. *Parallel Computing*, 52:78–105, 2016.

[59] Philipp Hupp, Riko Jacob, Mario Heene, Dirk Pflüger, and Markus Hegland. Global communication schemes for the sparse grid combination technique. In Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 564 – 573. IOS Press, 2014.

[60] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis De Supinski, and Rudolf Eigenmann. McrEngine: A scalable checkpointing system using data-aware aggregation and compression. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 17. IEEE Computer Society Press, 2012.

[61] David E Keyes. Exaflop/s: The why and the how. *Comptes Rendus Mécanique*, 339(2-3):70–77, 2011.

[62] Christoph Kowitz. *Applying the Sparse Grid Combination Technique in Linear Gyrokinetics*. PhD thesis, University of Munich, 2016.

[63] Christoph Kowitz and Markus Hegland. The sparse grid combination technique for computing eigenvalues in linear gyrokinetics. In *International Conference on Computational Science 2013*, Procedia Computer Science, pages 449–458, Amsterdam, June 2013. Elsevier.

[64] Christoph Kowitz and Markus Hegland. An opticom method for computing eigenpairs. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Munich 2012*, volume 97 of *Lecture Notes in Computational Science and Engineering*, pages 239–253. Springer International Publishing, 2014.

[65] Christoph Kowitz, Dirk Pflüger, Frank Jenko, and Markus Hegland. The combination technique for the initial value problem in linear gyrokinetics. In Michael Griebel and Jochen Garcke, editors, *Sparse Grids and Applications*, volume 88 of *Lecture Notes in Computational Science and Engineering*, pages 205–222, Heidelberg, October 2012. Springer.

[66] Christoph Joseph Kranz. *Untersuchungen zur Kombinationstechnik bei der numerischen Strömungssimulation auf versetzten Gittern*. PhD thesis, Technische Universität München, 2002.

[67] Jay Larson, Markus Hegland, Brendan Harding, Stephen Roberts, Linda Stals, Alistair Rendell, Peter Strazdins, Md Mohsin Ali, Christoph Kowitz, R Nobes, et al. Fault-tolerant grid-based solvers: Combining concepts from sparse grids and mapreduce. *Procedia Computer Science*, 18:130–139, 2013.

[68] Boris Lastdrager, Barry Koren, and Jan Verwer. *The Sparse-Grid Combination Technique Applied to Time-Dependent Advection Problems*, pages 143–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[69] Hermann Lederer, Reinhard Tisma, Roman Hatzky, Alberto Bottino, and Frank Jenko. Application enabling in deisa: Petascaling of plasma turbulence codes. In *PARCO*, pages 713–720, 2007.

[70] Daniel Marques, Keshav Pingali, Radu Rugina, and Sally A McKee. Compiler-enhanced incremental checkpointing for OpenMP applications. *PPoPP 2008*, page 275, 2008.

[71] F. Merz. *Gyrokinetic Simulation of Multimode Plasma Turbulence*. PhD thesis, Universität Münster, 2008.

[72] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In

Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11. IEEE Computer Society, 2010.

[73] Dirk Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, 2010.

[74] Dirk Pflüger. Spatially adaptive refinement. In Jochen Garcke and Michael Griebel, editors, *Sparse Grids and Applications*, Lecture Notes in Computational Science and Engineering, pages 243–262, Berlin Heidelberg, October 2012. Springer.

[75] Dirk Pflüger, Hans-Joachim Bungartz, Michael Griebel, Frank Jenko, Tilman Dannert, Mario Heene, Alfredo Parra Hinojosa, Christoph Kowitz, and Peter Zaspel. EXAHD: An exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*. Springer, September 2014.

[76] A. Rüttgers. *Multiscale Simulation of Polymeric Fluids using Sparse Grids*. PhD thesis, Institut für Numerische Simulation, Universität Bonn, 2016.

[77] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[78] Sergey A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. In *Dokl. Akad. Nauk SSSR*, volume 4, page 123, 1963.

[79] P. E. Strazdins, M. M. Ali, and B. Debusschere. Application fault tolerance for shrinking resources via the sparse grid combination technique. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1232–1238, May 2016.

[80] P. E. Strazdins, M. M. Ali, and B. Harding. Highly scalable algorithms for the sparse grid combination technique. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 941–950, May 2015.

[81] Peter Strazdins, Md Mohsin Ali, and Brendan Harding. Design and analysis of two highly scalable sparse grid combination algorithms. *Journal of Computational Science*, 17:547–561, 2016.

[82] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[83] D. Told, F. Jenko, J. M. TenBarge, G. G. Howes, and G. W. Hammett. Multiscale nature of the dissipation range in gyrokinetic simulations of alfvénic turbulence. *Phys. Rev. Lett.*, 115:025003, Jul 2015.

[84] Johannes Walter. Design and implementation of a fault simulation layer for the combination technique on HPC systems. Master's thesis, University of Stuttgart, 2016.

[85] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 524–533. IEEE, 2010.

[86] C. Zenger. Sparse grids. In W. Hackbusch, editor, *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 241 – 251. Vieweg, 1991.

[87] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.