Handling Silent Data Corruption with the Sparse Grid Combination Technique

Alfredo Parra Hinojosa, Brendan Harding, Markus Hegland, and Hans-Joachim Bungartz

Abstract We describe two algorithms to detect and filter silent data corruption (SDC) when solving time-dependent PDEs with the Sparse Grid Combination Technique (SGCT). The SGCT solves a PDE on many regular full grids of different resolutions, which are then combined to obtain a high quality solution. The algorithm can be parallelized and run on large HPC systems. We investigate silent data corruption and show that the SGCT can be used with minor modifications to filter corrupted data and obtain good results. We apply sanity checks before combining the solution fields to make sure that the data is not corrupted. These sanity checks are derived from well-known error bounds of the classical theory of the SGCT and do not rely on checksums or data replication. We apply our algorithms on a 2D advection equation and discuss the main advantages and drawbacks.

1 Introduction

Faults in high-end computing systems are now considered the norm rather than the exception [13]. The more complex these systems become, and the larger the number of components they have, the higher the frequency at which faults occur. Following the terminology in [33], a fault is simply the cause of an error. Errors, in turn, are categorized into three groups: (1) detected and corrected by hardware (DCE), (2) detected but uncorrectable errors (DUE), and (3) silent errors (SE). If an error leads to system failure, it is called *masked*; otherwise it is *unmasked*. We say that a system failed if there is a deviation from the correct service of a system function [2].

The field of fault tolerance explores ways to avoid system failures when faults occur. Different strategies can be followed depending on the type of fault. For example, one might be interested in tolerating the failure of single MPI processes,

A. Parra Hinojosa (✉) • H.-J. Bungartz
Technische Universität München, München, Germany
e-mail: hinojosa@in.tum.de; bungartz@in.tum.de

B. Harding • M. Hegland
Mathematical Sciences Institute, The Australian National University, Canberra, ACT, Australia
e-mail: brendan.harding@anu.edu.au; markus.hegland@anu.edu.au

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing – SPPEXA 2013-2015*, Lecture Notes in Computational Science and Engineering 113,
DOI 10.1007/978-3-319-40528-5_9

since one process failure can cause the whole application to crash, and new parallel libraries have been developed to handle these issues [5]. Another option is to use Checkpoint/Restart (C/R) algorithms, where the state of the simulation is stored to memory and retrieved in case of failure. The simulation is then restarted from the last complete checkpoint. Alternatively, developers could make replicas of certain critical processes as backups in case one of them fails [12].

These algorithms are usually applied when errors trigger a signal and thus can be easily detected. But we might run into problems if the errors don't trigger any signal. This is the case for *silent data corruption* (SDC), a common type of unmasked silent errors. SDC arises mainly in the form of undetected errors in arithmetic operations (most prominently as bit flips) and memory corruption [33]. Although SDC is expected to occur less often than detectable errors (such as hardware failure), one single occurrence of SDC could lead to entirely incorrect results [10]. The frequency at which silent errors occur has not been quantified rigorously, but evidence suggests that they occur frequently enough to be taken seriously [33].

Previous research has focused on algorithms that deal with detectable errors when using the SGCT [20]. We now want to understand the effect of SDC on the SGCT when solving PDEs. Elliott et al. [11] have outlined a methodology to model and simulate SDC, and they have described guidelines to design SDC-resilient algorithms. We adopt their recommendations in this paper, and we now briefly describe their main ideas.

1.1 Understanding Silent Data Corruption

Many algorithm designers start by assuming that SDC will occur exclusively in the form of bit flips. For this reason, they have chosen to simulate SDC by randomly injecting bit flips into an existing application, and then attempting to detect and correct wrong data. But this can only tell us how the application behaves in average, and one might fail to simulate the worst-case scenarios. Additionally, the exact causes of SDC in existing and future parallel systems are still poorly understood. For this reason we should avoid making assumptions about the exact causes of SDC. This lack of certainty does not mean that we should not attempt to simulate SDC (and, if possible, overcome it). On the contrary, by making no assumptions about the exact origins and types of SDC, we can focus on what really matters in terms of algorithm design: numerical errors. A robust algorithm should be able to handle numerical errors of arbitrary magnitude in the data without any knowledge of the specific sources of the error. In this way, the problem can be posed purely in terms of numerical analysis and error bounds.

But how does one actually design robust algorithms? There are several things to keep in mind. For instance, it is important to determine in which parts of the algorithm we cannot afford faults to occur, and in which parts we can relax this condition. This is called *selective reliability* [6]. It is also useful to identify *invariants* in a numerical algorithm. Energy conservation is a typical example, as

well as requiring a set of vectors to remain orthogonal. These invariants can be good places to start when searching for anomalous data. Furthermore, algorithm experts should try to develop cheap sanity checks to bound or exclude wrong results. Our research is largely based on this last recommendation. Finally, SDC can cause the control flow of the algorithm to deviate from its normal behavior. Although it is difficult to predict what this would mean for a specific application code, one can still turn to selective reliability to make sure that vulnerable sections of the code are dealt with properly by specifying conditions of correctness. We do not address problems in control flow explicitly in this work, but we do mention briefly how our algorithms could encompass this type of faults.

Many authors opt for a much more elaborate (and expensive) methodology based on checksums. (See [33], Sect. 5.4.2 for an extensive list of examples.) Implementing checksums even for a simple algorithm can prove a very difficult task. A good example is the self-healing, fault-tolerant preconditioned CG algorithm described in [8]. To perform the checksums, the authors require local diskless checkpointing, additional checkpoint processes, and a fault-tolerant MPI implementation. The programming effort and computational costs are substantial. In many cases checksums cannot be applied at all. Data replication can be sometimes useful (see [9]), but it also comes at a cost. These experiences motivate the search for new algorithmic, numerics-based solutions.

1.2 Statement of the Problem

We now want to translate these ideas into an SDC-resilient version of the Sparse Grid Combination Technique (SGCT) algorithm, which we describe in detail in the next section. The SGCT is a powerful algorithm that has been used to solve a wide variety of problems, from option pricing [31] and machine learning [14] to plasma physics [28] and quantum mechanics [16]. Our focus will be high-dimensional, time-dependent PDEs. Full grids with high discretization resolution are usually too computationally expensive, especially in higher dimensions. The SGCT solves the original PDE on different coarse, anisotropic full grids. Their coarseness makes them computationally cheap. The solutions on these coarse grids are then combined properly to approximate the full grid solution in an extrapolation-like manner. (We will see in Sect. 2.1 what it means to combine grids of different resolutions.) Our main concern is the following: the solution on one (or more) of the coarse grids might be wrong due to SDC, which can cause the final combined solution to be wrong as well. We therefore want to implement cheap sanity checks to make sure that wrong solutions are filtered and not considered for the combination. In a sense, the fact that the SGCT solves the same PDE on different grids means that it inherently shows data replication, and it is precisely this fact that we will exploit. But before continuing our discussion of SDC we take a small detour to recall the theory of sparse grids, and we describe the SGCT in detail.

2 Basics of Sparse Grids

Let us first introduce some notation. To discretize the unit interval $[0, 1]$ we use a one-dimensional grid Ω_l with $2^l - 1$ inner points and one point on each boundary ($2^l + 1$ points in total). This grid has mesh size $h_l := 2^{-l}$ and grid points $x_{l,j} := j \cdot h_l$ for $0 \leq j \leq 2^l$, with $l \in \mathbb{N} = \{1, 2, \ldots\}$.

In d dimensions we use underlined letters to denote multi-indices, $\mathbf{l} = (l_1, \ldots, l_d) \in \mathbb{N}^d$, and we discretize the d-unit cube using a d-dimensional full grid, $\Omega_\mathbf{l} := \Omega_{l_1} \times \cdots \times \Omega_{l_d}$. This grid has mesh sizes

$$h_\mathbf{l} := (h_{l_1}, \ldots, h_{l_d}) := 2^{-\mathbf{l}} \tag{1}$$

and grid points

$$x_{\mathbf{l},\mathbf{j}} := (x_{l_1,j_1}, \ldots, x_{l_d,j_d}) := \mathbf{j} \cdot h_\mathbf{l} \quad \text{for} \quad \mathbf{0} \leq \mathbf{j} \leq 2^\mathbf{l} \,. \tag{2}$$

Comparisons between multi-indices are done componentwise: two multi-indices $\mathbf{i}$ and $\mathbf{j}$ satisfy $\mathbf{i} \leq \mathbf{j}$ if $i_k \leq j_k$ for all $k \in \{1, \ldots, d\}$. (The same applies for similar operators.) We will also use discrete l_p-norms $|\cdot|_p$ for multi-indices. For example, $|\mathbf{l}|_1 := l_1 + \cdots + l_d$. Additionally, the operation $\mathbf{i} \wedge \mathbf{j}$ denotes the componentwise minimum of $\mathbf{i}$ and $\mathbf{j}$, i.e., $\mathbf{i} \wedge \mathbf{j} := (\min\{i_1, j_1\}, \ldots, \min\{i_d, j_d\})$. Finally, if $\mathscr{I}$ is a set of multi-indices, we define the *downset* of $\mathscr{I}$ as $\mathscr{I}_\downarrow := \{\mathbf{l} \in \mathbb{N}^d : \exists \mathbf{k} \in \mathscr{I} \text{ s.t. } \mathbf{l} \leq \mathbf{k}\}$. The downset $\mathscr{I}_\downarrow$ includes all multi-indices smaller or equal to all multi-indices in $\mathscr{I}$.

Suppose $u(\mathbf{x}) \in V \subset C([0, 1]^d)$ is the exact solution of a d-dimensional PDE. A numerical approximation of u will be denoted $u_\mathbf{i}(\mathbf{x}) \in V_\mathbf{i} \subset V$, where $V_\mathbf{i} = \bigotimes_{k=1}^d V_{i_k}$ is the space of piecewise d-linear functions defined on a grid $\Omega_\mathbf{i}$ [15],

$$V_\mathbf{i} := \operatorname{span}\{\phi_{\mathbf{i},\mathbf{j}} \,:\, \mathbf{0} \leq \mathbf{j} \leq 2^\mathbf{i}\} \,. \tag{3}$$

The d-dimensional hat functions $\phi_{\mathbf{i},\mathbf{j}}$ are the tensor product of one-dimensional hat functions,

$$\phi_{\mathbf{i},\mathbf{j}}(\mathbf{x}) := \prod_{k=1}^{d} \phi_{i_k,j_k}(x_k) \,, \tag{4}$$

with

$$\phi_{i,j}(x) := \max(1 - |2^i x - j|, 0) \,. \tag{5}$$

As a result, the interpolation of $u_\mathbf{i}(\mathbf{x})$ on grid $\Omega_\mathbf{i}$ can be written as

$$u_\mathbf{i}(\mathbf{x}) = \sum_{\mathbf{0} \leq \mathbf{j} \leq 2^\mathbf{i}} u_{\mathbf{i},\mathbf{j}} \phi_{\mathbf{i},\mathbf{j}}(\mathbf{x}) \,. \tag{6}$$

The coefficients $u_{\mathbf{i},\mathbf{j}} \in \mathbb{R}$ are simply the height of the hat functions $\phi_{\mathbf{i},\mathbf{j}}$ (see Fig. 1, left). We call (6) the *nodal representation* of $u_\mathbf{i}(\mathbf{x})$, and $u_{\mathbf{i},\mathbf{j}}$ are the *nodal coefficients*.

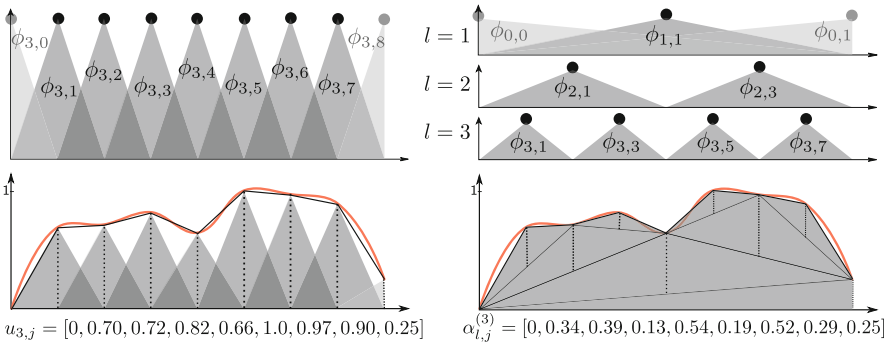

Fig. 1 Two different bases to interpolate a one-dimensional function using discretization level $i = 3$. *Left*: nodal representation. We store the values of $u_{i,j}$, which correspond to the height of the nodal hat functions. *Right*: hierarchical basis. We store the hierarchical coefficients $\alpha^{(i)}_{l,j}$, which represent the increments w.r.t. the previous level $l - 1$. Their magnitude decreases as the level l increases

Apart from $V_\mathbf{i}$ we will also consider *hierarchical spaces* $W_\mathbf{l}$ defined as

$$W_\mathbf{l} := \operatorname{span}\left\{\phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}) : \mathbf{j} \in \mathscr{I}_\mathbf{l}\right\} , \tag{7}$$

where the index set $\mathscr{I}_\mathbf{l}$ is given by

$$\mathscr{I}_\mathbf{l} := \left\{\mathbf{j} : 1 \leq j_k \leq 2^{l_k} - 1, j_k \text{ odd}, 1 \leq k \leq d\right\} . \tag{8}$$

A hierarchical space $W_\mathbf{i}$ can be defined as the space of all functions $u_\mathbf{i} \in V_\mathbf{i}$ such that $u_\mathbf{i}$ is zero on all grid points in the set $\bigcup_{\mathbf{l}<\mathbf{i}} \Omega_\mathbf{l}$ [20]. The space $W_\mathbf{1}$ is treated separately, since it is endowed with two additional basis functions $\phi_{0,0}$ and $\phi_{0,1}$ to include the boundary conditions, as illustrated in Fig. 1.[1] Using hierarchical spaces allows us to decompose a space $V_\mathbf{i}$ as

$$V_\mathbf{i} = \bigoplus_{\mathbf{l}\leq\mathbf{i}} W_\mathbf{l} . \tag{9}$$

Equations (7), (8), and (9) tell us that each $u_\mathbf{i} \in V_\mathbf{i}$ can be written alternatively as

$$u_\mathbf{i}(\mathbf{x}) = \sum_{\mathbf{l}\leq\mathbf{i}} h_\mathbf{l}(\mathbf{x}), \quad h_\mathbf{l}(\mathbf{x}) \in W_\mathbf{l} \tag{10}$$

$$= \sum_{\mathbf{l}\leq\mathbf{i}} \sum_{\mathbf{j}\in\mathscr{I}_\mathbf{l}} \alpha^{(\mathbf{i})}_{\mathbf{l},\mathbf{j}} \phi_{\mathbf{l},\mathbf{j}}(\mathbf{x}) . \tag{11}$$

[1] For a detailed discussion on the boundary treatment, see [30].

This is the *hierarchical representation* of $u_{\mathbf{i}}(\mathbf{x})$, and $\alpha_{\mathbf{l,j}}^{(\mathbf{i})} \in \mathbb{R}$ are the *hierarchical coefficients* or *hierarchical surpluses*. This decomposition is illustrated for a 1D function in Fig. 1 (right). The hierarchical coefficients can be directly obtained from the values of $u_{\mathbf{i}}$ at the corresponding grid points. In one dimension, they are calculated as

$$\begin{aligned} \alpha_{l,j}^{(i)} &= u_i(x_{l,j}) - \frac{1}{2}(u_i(x_{l,j-1}) + u_i(x_{l,j+1})) \\ &= \left[-\tfrac{1}{2} \quad 1 \quad -\tfrac{1}{2}\right]_{l,j} u_i(x_{l,j}) \ . \end{aligned} \tag{12}$$

This operation is called *hierarchization*, and can be extended to d dimensions using the one-dimensional stencil above,

$$\alpha_{\mathbf{l,j}}^{(\mathbf{i})} = \left(\prod_{k=1}^{d} \left[-\tfrac{1}{2} \quad 1 \quad -\tfrac{1}{2}\right]_{l_k,j_k} \right) u_{\mathbf{i}}(x_{\mathbf{l,j}}) \ . \tag{13}$$

This is simply a transformation from the nodal to the hierarchical basis. The inverse operation (calculating the nodal coefficients from the hierarchical coefficients) is called *dehierarchization*.

If we discretize a problem on a uniform d-dimensional full grid $\Omega_{\mathbf{n}}$ (with mesh size $h_n = 2^{-n}$ in every dimension), this grid will have $\mathcal{O}(h_n^{-d}) = \mathcal{O}(2^{nd})$ grid points. This exponential dependence on n and d makes running algorithms on such grids infeasible, a fact commonly referred to as the *curse of dimensionality*. *Sparse grids* aim to alleviate the curse of dimensionality via a hierarchical approach [7, 15]. The classical sparse grid space $V_n^s \subset V_n$ is defined as

$$V_n^s := \bigoplus_{|\mathbf{l}|_1 \leq n+d-1} W_{\mathbf{l}} \ , \tag{14}$$

which we have illustrated in Fig. 2 for $d = 2$ and $n = 4$. A sparse grid has $\mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1})$ points, which represents a dramatic reduction from the $\mathcal{O}(h_n^{-d})$

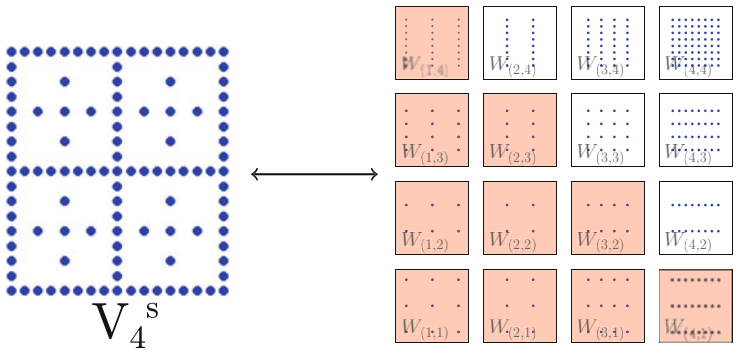

Fig. 2 A sparse grid of level 4 and the hierarchical subspaces that compose it

discretization points required by a full grid of the same level n. However, for functions whose mixed second derivatives are bounded, the interpolation error on a sparse grid is in $\mathcal{O}(h_n^2(\log h_n^{-1})^{d-1})$, only slightly larger than on a full grid, which is in $\mathcal{O}(h_n^2)$ [30].

2.1 The Sparse Grid Combination Technique

Sparse grids allow us to reduce the number of degrees of freedom in a discrete problem without sacrificing much in terms of accuracy. However, it is in general difficult to discretize a problem on a sparse grid. Luckily, there exists a variant of sparse grids, the *Sparse Grid Combination Technique* (SGCT)[17, 18], which can overcome this problem. We illustrate the SGCT using a simple time-dependent PDE, the linear advection equation in two dimensions plus time given by

$$\frac{\partial u}{\partial t} + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} = 0 \,, \tag{15}$$

in the unit square $(x, y) \in [0, 1]^2$ with initial condition $u(x, y, t = 0) = \sin(2\pi x)\sin(2\pi y)$ and periodic boundary conditions in x and y. The velocities c_x and c_y are real positive constants. The analytical solution of (15) is $u(x, y, t) = \sin(2\pi(x - c_x t))\sin(2\pi(y - c_y t))$.

Suppose we use an explicit discretization scheme in time, such as Lax-Wendroff [34]. In Fig. 3 (left) we have plotted the solution of (15) at time $t = 0.5$ with $c_x = c_y = 0.5$ using the Lax-Wendroff scheme on a grid $\Omega_{\mathbf{n}}$ of discretization level $\mathbf{n} = (4, 4)$ (i.e. with $(2^4 + 1) \times (2^4 + 1)$ points). On the right, we have done the same but on five coarser grids of different discretization level, each of which has four times fewer discretization points than the grid $\Omega_{(4,4)}$. The idea behind the SGCT is to *combine* those five grids with weights $+1$ and -1 (as indicated in the

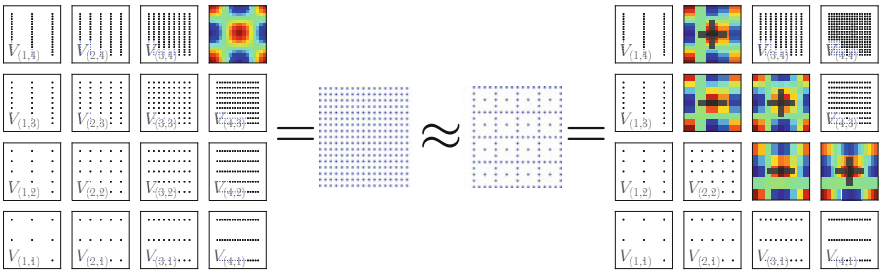

Fig. 3 A classical SGCT to solve the 2D advection equation using $\mathbf{n} = (4, 4)$ and $\tau = 2$ ($\mathbf{i}_{\min} = (2, 2)$). The combination of the five resulting grids results in a sparse grid approximation of the full grid $\Omega_{(4,4)}$

figure) to obtain an approximation of the solution on grid $\Omega_{(4,4)}$. The union of these grids results in a sparse grid, depicted on the middle right.

The grids are usually combined either using interpolation or hierarchization. The former means that each combination grid is interpolated to the full grid space (in our example, $\Omega_{(4,4)}$) and the grids are combined together in this space. The latter requires us to transform each solution into the hierarchical basis (using Eq. (13)), after which the hierarchical coefficients $\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{i})}$ can be added directly in the sparse grid space [23, 25].

We can now write down the classical definition of the SGCT. We approximate the full grid solution $u_{\mathbf{n}}$ by a function $u_{\mathbf{n}}^{(c)}$ as follows:

$$u_{\mathbf{n}} \approx u_{\mathbf{n}}^{(c)} = \sum_{\mathbf{i} \in \mathscr{I}} c_{\mathbf{i}} u_{\mathbf{i}} \,. \tag{16}$$

The weights $c_{\mathbf{i}} \in \mathbb{R}$ are called *combination coefficients*, and each $u_{\mathbf{i}}$ is a numerical approximation of u on a coarse, anisotropic full grid $\Omega_{\mathbf{i}}$. We call the $u_{\mathbf{i}}$ *combination solutions*, the grids $\Omega_{\mathbf{i}}$ *combination grids*, $u_{\mathbf{n}}^{(c)}$ the *combined solution*, and its corresponding grid $\Omega_{\mathbf{n}}^{(c)}$ the *combined grid* (which is a sparse grid). $\mathscr{I}$ is a set of multi-indices.

The approximation quality of the combined solution (16) strongly depends on the coefficients $c_{\mathbf{i}}$ and the set $\mathscr{I}$, since only certain choices yield reasonable results. One such choice is the *classical combination technique* given by

$$u_{\mathbf{n}}^{(c)} = \underbrace{\sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q}}_{=c_{\mathbf{i}}} \sum_{\mathbf{i} \in \mathscr{I}_{\mathbf{n},q,\tau}} u_{\mathbf{i}} \,. \tag{17}$$

Here, the index set $\mathscr{I}_{\mathbf{n},q,\tau}$ is defined as

$$\mathscr{I}_{\mathbf{n},q,\tau} := \{\mathbf{i} \in \mathbb{N}^d : |\mathbf{i}|_1 = |\mathbf{i}_{\min}|_1 + \tau - q \quad \text{and} \quad \mathbf{i}_{\min} = \mathbf{n} - \tau \cdot \mathbf{1} > \mathbf{0}\} \,, \tag{18}$$

where $\tau \in \mathbb{N}$, and $\mathbf{i}_{\min}$ specifies a minimal resolution level in each dimension. The classical combination technique depicted in Fig. 3 was generated by choosing $\mathbf{n} = (4,4)$ and $\tau = 2$ ($\mathbf{i}_{\min} = (2,2)$), giving the combination

$$u_{\mathbf{n}}^{(c)} = u_{(2,4)} + u_{(3,3)} + u_{(4,2)} - u_{(2,3)} - u_{(3,2)} \,.$$

For a general combination of the form (16), the combination coefficients can be calculated as

$$c_{\mathbf{i}} = \sum_{\mathbf{i} \le \mathbf{j} \le \mathbf{i}+\mathbf{1}} (-1)^{|\mathbf{j}-\mathbf{i}|} \chi_{\mathscr{I}}(\mathbf{j}) \,, \tag{19}$$

where $\chi_{\mathscr{I}}$ is the indicator function of set $\mathscr{I}$ [19].

The main advantage of the SGCT is that it approximates a full grid solution very well by using a combination of solutions on coarse anisotropic grids. The combination of these grids results in a sparse grid, but we avoid discretizing our problem directly on the sparse grid, which is cumbersome and requires complex data structures. However, by using the combination technique we have an extra storage overhead compared to a single sparse grid, since there is some data redundancy among the combination grids. The storage requirements of the combination technique are of order $\mathcal{O}(d(\log h_n^{-1})^{d-1}) \times \mathcal{O}(h_n^{-1})$ [30]. This data redundancy is the key feature of the SGCT that will allow us to deal with data corruption.

Finally, it is important to mention that the combination coefficients $c_{\mathbf{i}}$ and the index set $\mathscr{I}$ can be chosen in various different ways, and the resulting combinations vary in approximation quality. This is the underlying idea behind dimension-adaptive sparse grids [24], and their construction is inherently fault tolerant.

3 The SGCT in Parallel and Fault Tolerance with the Combination Technique

The SGCT offers two levels of parallelism. First, since we solve the same PDE on different grids, each solver call is independent of the rest. Second, on each grid one can use domain decomposition (or other parallelization techniques). But if the PDE is time-dependent we have to combine the solutions every certain number of time steps to avoid divergence, which requires communication. The fraction of time spent in the solver and in the communication steps depends on how often one combines the grids. If the PDE is not time-dependent we combine only once at the end. Algorithm 2 summarizes the main components of a parallel implementation of the classical SGCT. It can be implemented using a master/slave scheme. The master distributes the work and coordinates the combination of the grids. The slaves solve the PDE on the different grids and communicate the results to the master. Most of

Algorithm 2 Classical SGCT in Parallel

input: A function SOLVER; maximum resolution $\mathbf{n}$; parameter τ; time steps per combination N_t
output: Combined solution $u_{\mathbf{n}}^{(c)}$
Generate index set $\mathscr{I}_{\mathbf{n},q,\tau}$ ▷ Eq. (18)
Calculate combination coefficients $c_{\mathbf{i}}$ ▷ Eq. (19)
for $\mathbf{i} \in \mathscr{I}_{\mathbf{n},q,\tau}$ **do**
    $u_{\mathbf{i}} \leftarrow u(\mathbf{x}, t = 0)$ ▷ Set initial conditions by sampling
while not converged **do**
    for $\mathbf{i} \in \mathscr{I}_{\mathbf{n},q,\tau}$ **do in parallel**
        $u_{\mathbf{i}} \leftarrow$ SOLVER($u_{\mathbf{i}}$,N_t) ▷ Solve the PDE on grid $\Omega_{\mathbf{i}}$ (N_t time steps)
        $u_{\mathbf{i}} \leftarrow$ HIERARCHIZE($u_{\mathbf{i}}$) ▷ Transform to hier. basis, Eq. (13)
    $u_{\mathbf{n}}^{(c)} \leftarrow$ REDUCE($c_{\mathbf{i}}$ $u_{\mathbf{i}}$) ▷ Combined solution (in the hier. basis)
    $u_{\mathbf{n}}^{(c)} \leftarrow$ DEHIERARCHIZE($u_{\mathbf{n}}^{(c)}$) ▷ Transform back to nodal basis
    for $\mathbf{i} \in \mathscr{I}_{\mathbf{n},q,\tau}$ **do**
        $u_{\mathbf{i}} \leftarrow$ SCATTER($u_{\mathbf{n}}^{(c)}$) ▷ Sample each $u_{\mathbf{i}}$ from new $u_{\mathbf{n}}^{(c)}$

the time and computational resources are spent on the calls to the actual PDE solver, line 9.

We are currently working on a massively parallel implementation of the SGCT [22]. It is a *C++* framework that can call existing PDE solvers (e.g. *DUNE* [3]) and apply the SGCT around them. To develop such an environment, three major issues have to be carefully taken into consideration:

1. *Load balancing*. The work load (calculating all combination solutions $u_{\mathbf{i}}$, line 9) has to be distributed properly among the computing nodes. The time to solution of each $u_{\mathbf{i}}$ depends on the number of unknowns and the anisotropy of each grid $\Omega_{\mathbf{i}}$. This problem has been studied in [21].
2. *Communication*. After performing N_t time steps, the different $u_{\mathbf{i}}$ have to be combined, which requires communication (line 11). The combined solution $u_{\mathbf{n}}^{(c)}$ is used as initial condition for the next N_t time steps for all combination solutions $u_{\mathbf{i}}$ (line 14), and this also requires communication. Efficient communication patterns for the SGCT have been studied in detail in [26]. The problem of determining how often the grids should be combined is still under investigation, since the frequency depends on the specific PDE.
3. *Fault tolerance*. In light of increasing hardware and software faults, the *Fault Tolerant Combination Technique* (FTCT) has been developed [20]. It has been applied to plasma physics simulations and proved to scale well when *hard faults* occur [1, 29]. This is the area where our group contributes to the *C++* framework.

All three points raise interesting algorithmic questions, but since the third plays a central role in our discussion of SDC, we should add a few words about it. In Fig. 4 (left) we depict a classical SGCT with $\mathbf{n} = (5, 5)$ and $\tau = 3$ ($\mathbf{i}_{\min} = (2, 2)$). Suppose the system encounters a fault during the call to `solver`. As a result, one or more combination solutions $u_{\mathbf{i}}$ will be lost. In Fig. 4 (left) we have assumed that solution $u_{(4,3)}$ has been lost due to a fault. Instead of recomputing this lost solution, the FTCT attempts to find alternative ways of combining the successfully calculated solutions, excluding the solutions lost due to faults. The possible alternative combinations are almost as good as the original combination. In Fig. 4 (right) we see an alternative combination that excludes solution $u_{(4,3)}$, using instead solution $u_{(3,2)}$. But notice that $u_{(3,2)}$ was not part of the original set of solutions. The main idea of the FTCT is to compute some extra solutions beforehand (such as $u_{(3,2)}$) and use them only in case of faults. This results in a small extra overhead, but it has been shown to scale [20]. In the original SGCT we solve the PDE on grids $\Omega_{\mathbf{i}}$ with index $\mathbf{i} \in \mathscr{I}_{\mathbf{n},q,\tau}$ for $q = 0, \ldots, d-1$, Eq. (17). The FTCT extends this set to include the indices that result from setting $q = d, d+1$. We call this set $\mathscr{I}^{\text{ext}}_{\mathbf{n},q,\tau}$. The combination coefficients $c_{\mathbf{i}}$ that correspond to these extra solutions are set to zero if no faults occur and can become nonzero if faults occur.

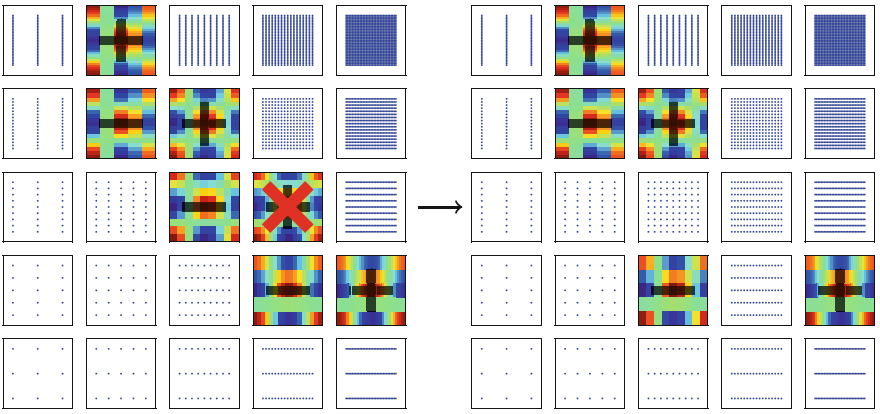

Fig. 4 Example of FTCT with $\mathbf{n} = (5, 5)$ and $\tau = 3$. The original index set $\mathscr{I}_{(5,5),q,3}$ is extended with the additional sets $\mathscr{I}_{(5,5),2,3} = \{(2, 3), (3, 2)\}$ and $\mathscr{I}_{(5,5),3,3} = \{(2, 2)\}$, and the additional solutions are used only in case of faults

3.1 SDC and the Combination Technique

If SDC occurs at some stage of Algorithm 2, it is most likely to happen during the call to the solver, line 9, which is where most time is spent. This means that the solver could return a wrong answer, and this would taint the combined solution $u_{\mathbf{n}}^{(c)}$ during the combination (reduce) step in line 11. Since the combination step is a linear operation, the error introduced in the combined solution would be of the same magnitude of the affected combination solution. Additionally, if the convergence criterion in line 7 has not been met, the scatter step would propagate the spurious data to all other combination solutions, potentially ruining the whole simulation.

To simulate an occurrence of SDC we follow an approach similar to [10]. Let us first assume that SDC only affects one combination solution $u_{\mathbf{i}}$, and that only one of its values $u_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}})$ is altered in one of the following ways:

1. $\tilde{u}_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}}) = u_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}}) \times 10^{+5}$ (very large)[2]
2. $\tilde{u}_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}}) = u_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}}) \times 10^{-0.5}$ (slightly smaller)
3. $\tilde{u}_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}}) = u_{\mathbf{i}}(x_{\mathbf{i},\mathbf{j}}) \times 10^{-300}$ (very small)

This fault injection is performed only once throughout the simulation,[3] at a given time step. Altering only one value of one combination solution is a worst-case scenario, since a solution with many wrong values should be easier to detect. If the wrong solution is not detected, the wrong data would propagate to other grids during

[2]The authors in [10] use a factor of 10^{+150} to cover all possible orders of magnitude, but we choose 10^{+5} simply to keep the axes of our error plots visible. The results are equally valid for 10^{+150}.

[3]The assumption that SDC occurs only once in the simulation is explained in [10].

the scatter step. In our tests we also simulated additive errors of various magnitudes, as well as random noise, and the results did not offer new insights. We thus focus on multiplicative errors in what follows.

If we want to make sure that all $u_{\mathbf{i}}$ have been computed correctly, we should introduce *sanity checks* before the combination step (between lines 10 and 11). These checks should not be problem-dependent, since the function *solver* could call any arbitrary code. Although it is not possible in general to know if a given combination solution $u_{\mathbf{i}}$ is correct, *we have many of them* (typically tens or hundreds), each with a different discretization resolution. We can therefore use this redundant information to determine if one or more solutions have been affected by SDC. We know that all $u_{\mathbf{i}}$ should look similar, since they are all solutions of the same PDE. The question is how similar? Or equivalently, how *different* can we expect an arbitrary pair of combination solutions (say $u_{\mathbf{s}}$ and $u_{\mathbf{t}}$) to be from each other? If two solutions look somehow different we can ask if such a difference falls within what is theoretically expected or not. The theory of the SGCT provides a possible answer to this question.

Early studies of the SGCT show that convergence can be guaranteed if each $u_{\mathbf{i}}$ satisfies the *error splitting assumption* (ESA) [18], which for arbitrary dimensions can be written as [19]

$$u - u_{\mathbf{i}} = \sum_{k=1}^{d} \sum_{\substack{\{e_1,\dots,e_k\} \\ \subset\{1,\dots,d\}}} C_{e_1,\dots,e_k}(\mathbf{x}, h_{i_{e_1}}, \dots, h_{i_{e_k}}) h_{i_{e_1}}^p \cdots h_{i_{e_k}}^p \ , \tag{20}$$

where $p \in \mathbb{N}$ and each function $C_{e_1,\dots,e_k}(\mathbf{x}, h_{i_{e_1}}, \dots, h_{i_{e_k}})$ depends on the coordinates $\mathbf{x}$ and on the different mesh sizes $h_{\mathbf{i}}$. Additionally, for each $\{e_1, \dots, e_k\} \subset \{1, \dots, d\}$ one has $|C_{e_1,\dots,e_k}(\mathbf{x}, h_{i_{e_1}}, \dots, h_{i_{e_k}})| \leq \kappa_{e_1,\dots,e_k}(\mathbf{x})$, and all $\kappa_{e_1,\dots,e_k}$ are bounded by $\kappa_{e_1,\dots,e_k}(\mathbf{x}) \leq \kappa(\mathbf{x})$. Equation (20) is a *pointwise* relation, i.e. it must hold for all points $\mathbf{x}$ independently, which can be seen by the explicit dependence of each function $C_{e_1,\dots,e_k}$ on $\mathbf{x}$.

In one dimension ($d = 1$), the ESA is simply

$$u - u_i = C_1(x_1, h_i) h_i^p, \quad |C_1(x_1, h_i)| \leq \kappa_1(x_1) \ . \tag{21}$$

From (12) it follows that the hierarchical coefficients also satisfy the ESA

$$\alpha_{l,j} - \alpha_{l,j}^{(i)} = D_1(x_{l,j}, h_i) h_i^p, \quad |D_1(x_{l,j}, h_i)| \leq 2\kappa_1(x_{l,j}) \ , \tag{22}$$

where $\alpha_{l,j}$ is the exact hierarchical coefficient at point $x_{l,j}$.

Similarly, in two dimensions we have

$$u - u_{\mathbf{i}} = C_1(x_1, x_2, h_{i_1}) h_{i_1}^p + C_2(x_1, x_2, h_{i_2}) h_{i_2}^p + C_{1,2}(x_1, x_2, h_{i_1}, h_{i_2}) h_{i_1}^p h_{i_2}^p \ . \tag{23}$$

There are univariate contributions from each dimension and a cross term that depends on both dimensions. Analogously, for the hierarchical coefficients we have

$$\alpha_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{i})}_{\mathbf{l},\mathbf{j}} = D_1(x_{\mathbf{l},\mathbf{j}}, h_{i_1})h^p_{i_1} + D_2(x_{\mathbf{l},\mathbf{j}}, h_{i_2})h^p_{i_2} + D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{i_1}, h_{i_2})h^p_{i_1}h^p_{i_2}\,, \tag{24}$$

with $|D_1| \leq 4\kappa_1(x_{\mathbf{l},\mathbf{j}})$, $|D_2| \leq 4\kappa_2(x_{\mathbf{l},\mathbf{j}})$, and $|D_{1,2}| \leq 4\kappa_{1,2}(x_{\mathbf{l},\mathbf{j}})$. This follows from (13).

Now suppose we take two arbitrary combination solutions $u_{\mathbf{s}}$ and $u_{\mathbf{t}}$ in two dimensions. If these two solutions satisfy the ESA it is straightforward to show that the difference of their corresponding hierarchical coefficients satisfies

$$\begin{aligned}\alpha^{(\mathbf{t})}_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{s})}_{\mathbf{l},\mathbf{j}} = {} & D_1(x_{\mathbf{l},\mathbf{j}}, h_{t_1})h^p_{t_1} + D_2(x_{\mathbf{l},\mathbf{j}}, h_{t_2})h^p_{t_2} + D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{t_1}, h_{t_2})h^p_{t_1}h^p_{t_2} \\ & - D_1(x_{\mathbf{l},\mathbf{j}}, h_{s_1})h^p_{s_1} - D_2(x_{\mathbf{l},\mathbf{j}}, h_{s_2})h^p_{s_2} - D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{s_1}, h_{s_2})h^p_{s_1}h^p_{s_2}\,.\end{aligned} \tag{25}$$

Clearly, this equation holds only for the hierarchical spaces common to both grids $\Omega_{\mathbf{s}}$ and $\Omega_{\mathbf{t}}$, i.e. for all $W_{\mathbf{l}}$ with $(1,1) \leq \mathbf{l} \leq \mathbf{s} \wedge \mathbf{t}$. Equation (25) tells us that the difference between the hierarchical coefficients of two combination solutions depends mainly on two things: (1) how coarse or fine the grids are, and (2) the distance $|\mathbf{t} - \mathbf{s}|_1$, which tells us whether grids $\Omega_{\mathbf{s}}$ and $\Omega_{\mathbf{t}}$ have similar discretization resolutions. The former can be observed by the explicit dependence on $h_{\mathbf{t}}$ and $h_{\mathbf{s}}$, dominated by the univariate terms. The latter means that if two grids have similar discretizations, the terms in (25) will tend to cancel each other out. Equation (25) is (pointwise) bounded by

$$\beta^{(\mathbf{s},\mathbf{t})}_{\mathbf{l},\mathbf{j}} := |\alpha^{(\mathbf{t})}_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{s})}_{\mathbf{l},\mathbf{j}}| \leq 4 \cdot \kappa(x_{\mathbf{l},\mathbf{j}}) \cdot (h^p_{t_1} + h^p_{s_1} + h^p_{t_2} + h^p_{s_2} + h^p_{t_1}h^p_{t_2} + h^p_{s_1}h^p_{s_2}), \quad \mathbf{l} \leq \mathbf{s} \wedge \mathbf{t}\,. \tag{26}$$

This result can help us to detect SDC, as we soon show.

Our goal is to implement Algorithm 3. A sanity check is done before the combination step (line 11). This is where we attempt to detect and filter wrong

Algorithm 3 FTCT with sanity checks for SDC

1: **input:** A function SOLVER; maximum resolution $\mathbf{n}$; parameter τ; time steps per combination N_t
2: **output:** Combined solution $u^{(c)}_{\mathbf{n}}$
3: Generate extended index set $\mathscr{I}^{ext}_{\mathbf{n},q,\tau}$
4: Calculate combination coefficients $c_{\mathbf{i}}$ ▷ Eq. (19)
5: **for** $\mathbf{i} \in \mathscr{I}^{ext}_{\mathbf{n},q,\tau}$ **do in parallel**
6: $u_{\mathbf{i}} \leftarrow u(\mathbf{x}, t = 0)$ ▷ Set initial conditions by sampling
7: **while** not converged **do**
8: **for** $\mathbf{i} \in \mathscr{I}^{ext}_{\mathbf{n},q,\tau}$ **do in parallel**
9: $u_{\mathbf{i}} \leftarrow$ SOLVER($u_{\mathbf{i}}$,N_t) ▷ Solve the PDE on grid $\Omega_{\mathbf{i}}$ (N_t time steps)
10: $u_{\mathbf{i}} \leftarrow$ HIERARCHIZE($u_{\mathbf{i}}$) ▷ Transform to hier. basis, Eq. (13)
11: $\{\mathbf{i}_{\text{sdc}}\} \leftarrow$ SDCSANITYCHECK($\{u_{\mathbf{i}}\}$) ▷ Check for SDC in all $u_{\mathbf{i}}$
12: **if** $\{\mathbf{i}_{\text{sdc}}\}$ not empty **then** ▷ Did SDC affect any $u_{\mathbf{i}}$?
13: $\{c_{\mathbf{i}}\} \leftarrow$ COMPUTENEWCOEFFS($\{\mathbf{i}_{\text{sdc}}\}$) ▷ Update combination coeffs.
14: $u^{(c)}_{\mathbf{n}} \leftarrow$ REDUCE($c_{\mathbf{i}}$ $u_{\mathbf{i}}$) ▷ Combined solution (in the hier. basis)
15: $u^{(c)}_{\mathbf{n}} \leftarrow$ DEHIERARCHIZE($u^{(c)}_{\mathbf{n}}$) ▷ Transform back to nodal basis
16: **for** $\mathbf{i} \in \mathscr{I}^{ext}_{\mathbf{n},q,\tau}$ **do**
17: $u_{\mathbf{i}} \leftarrow$ SCATTER($u^{(c)}_{\mathbf{n}}$) ▷ Sample each $u_{\mathbf{i}}$ from new $u^{(c)}_{\mathbf{n}}$

combination solutions, based on (26). If we are able to detect whether one or more combination solutions are wrong, we can apply the FTCT, treating wrong solutions in the same way as when hard faults occur, finding a new combination of solutions that excludes them, see Fig. 4. We now describe two possible implementations of the function `sdcSanityCheck`.

3.2 Sanity Check 1: Filtering SDC via Comparison of Pairs of Solutions

The first possible implementation of a simple sanity check is to apply (26) directly: we compare pairs of solutions $u_{\mathbf{s}}$ and $u_{\mathbf{t}}$ in their hierarchical basis and make sure that the bound (26) is fulfilled. If one of the solutions is wrong due to SDC, the quantity $\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ will be large and the bound might not be fulfilled, indicating that something is wrong. Unfortunately, the constant $\kappa(x_{\mathbf{l},\mathbf{j}})$ in the bound is in fact a function of space and is problem-dependent. This means that it has to be approximated somehow at all points $x_{\mathbf{l},\mathbf{j}}$, which is not trivial. It is only possible to estimate it once the solutions $u_{\mathbf{i}}$ have been calculated, but this is done assuming that all $u_{\mathbf{i}}$ have been computed correctly. Of course, this assumption does not hold if SDC can occur.

Despite these disadvantages, it is still possible to use bound (26) to detect and filter SDC. First, note that the function $\kappa(x_{\mathbf{l},\mathbf{j}})$ decays exponentially with increasing level $\mathbf{l}$. This is due to the fact that the hierarchical coefficients $\alpha_{\mathbf{l},\mathbf{j}}$ themselves decay exponentially with $\mathbf{l}$ (see Fig. 1, right). For a simple interpolation problem they behave as [7]

$$|\alpha_{\mathbf{l},\mathbf{j}}| \leq 2^{-d} \cdot \left(\frac{2}{3}\right)^{d/2} \cdot 2^{-(3/2)\cdot|\mathbf{l}|_1} \cdot \left\| D^{\mathbf{2}}(u|_{\operatorname{supp}\phi_{\mathbf{l},\mathbf{j}}}) \right\|_{L_2} , \tag{27}$$

with $D^{\mathbf{2}}(u) := \frac{\partial^4 u}{\partial x_1^2 \partial x_2^2}$. We can account for this exponential decay by normalizing the quantity $\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ as follows:

$$\hat{\beta}_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})} := \frac{|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|}{\min\left\{|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}|, |\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|\right\}} \quad \text{for all } \mathbf{l} \leq \mathbf{s} \wedge \mathbf{t}, \quad \mathbf{0} \leq \mathbf{j} \leq 2^{\mathbf{l}} . \tag{28}$$

If no SDC occurs, $|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|$ and $|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}|$ should be very similar, so it does not matter which of the two we use for the normalization. But if SDC occurs and their difference is large, dividing by the smaller one will amplify this difference. We can then take the largest $\hat{\beta}_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ over all grid points $x_{\mathbf{l},\mathbf{j}}$,

$$\hat{\beta}^{(\mathbf{s},\mathbf{t})} := \max_{\mathbf{l} \leq \mathbf{s} \wedge \mathbf{t}} \max_{\mathbf{j} \in \mathscr{I}_{\mathbf{l}}} \hat{\beta}_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})} . \tag{29}$$

Algorithm 4 Sanity check via comparison of solutions

1: **input:** The set of all combination solutions $\{u_{\mathbf{i}}\}$ (in the hierarchical basis)
2: **output:** The set of indices corresponding to the solutions affected by SDC, $\{\mathbf{i}_{\text{sdc}}\}$
3: **function** SDCSANITYCHECK($\{u_{\mathbf{i}}\}$)
4: **for** all pairs $(u_{\mathbf{s}}, u_{\mathbf{t}})$ with $\mathbf{s}, \mathbf{t} \in \mathscr{I}^{ext}_{\mathbf{n},q,\tau}$ **do**
5: Compute $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ ▷ Eq. (29)
6: **if** $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ too large **then**
7: Mark pair $(\mathbf{s}, \mathbf{t})$ as corrupted
8: From list of corrupted pairs $(\mathbf{s}, \mathbf{t})$, determine corrupted grids $\{\mathbf{i}_{\text{sdc}}\}$
9:
10: Return$\{\mathbf{i}_{\text{sdc}}\}$
11:

Pair	$\hat{\beta}^{(\mathbf{s},\mathbf{t})}$
(7, 9) (7, 8)	**8.71e+04**
(7, 8) (9, 7)	**4.95e+04**
(8, 7) (7, 7)	2.50e-02
(7, 9) (8, 8)	3.67e-02
(7, 7) (8, 8)	4.91e-02
(8, 7) (8, 8)	2.50e-02
(7, 9) (8, 7)	6.03e-02
(7, 9) (7, 7)	3.76e-02
(9, 7) (7, 7)	3.76e-02
(9, 7) (8, 8)	3.67e-02
(7, 8) (7, 7)	**5.01e+04**
(8, 7) (7, 8)	**4.97e+04**
(8, 7) (9, 7)	1.24e-02
(7, 9) (9, 7)	7.24e-02
(7, 8) (8, 8)	**8.68e+04**

This quantity simply gives us the largest (normalized) difference between two combination solutions in the hierarchical basis, and it does not decay exponentially in $\mathbf{l}$. Our goal is to keep track of this quantity, expecting it to be small for all pairs of combination solutions. If this is not the case for a specific pair $(u_{\mathbf{s}}, u_{\mathbf{t}})$ we can conclude that one solution (or both) was not computed correctly during the call to the function `solver`.

Algorithm 4 summarizes this possible implementation of the function `sdcSanityCheck`. The table on the right illustrates what the function generates for a simple implementation of a 2D FTCT. We solved once again the advection equation (15) for $t = 0.25$ and 129 time steps, with $c_x = c_y = 0.5$. The FTCT parameters used where $\mathbf{n} = (9, 9)$ and $\tau = 2$, which results in six combination grids. We injected SDC of small magnitude (case 2 from the previous section) into one of the combination grids at the very last time step. The table shows a list of all pairs $(\mathbf{s}, \mathbf{t})$ and the calculated value of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$. Some pairs have unusually large values of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$, shown in boldface. It should be evident that solution $u_{(7,8)}$ has been affected by SDC, being the only one appearing in all five pairs marked as corrupted.

There remains one unanswered question: which values of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ should be considered and marked as "too large" (lines 7 and 8)? We discussed that it is difficult to calculate a specific value for the upper bound (26), since $\kappa(x_{\mathbf{l},\mathbf{j}})$ is problem-dependent. But we might not need to. We simply need to recognize that some values of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ are disproportionately large *compared to the rest*. The values highlighted in the table are indeed clear outliers, and the wrong solution can be identified. The idea of detecting outliers leads us to our second implementation of `sdcSanityCheck`.

3.3 Sanity Check 2: Filtering SDC via Outlier Detection

So far we have used two facts about the combination technique to be able to deal with SDC. First, although we cannot tell in general if one single combination solution has been computed correctly, we know that all combination solutions should look somewhat similar. And second, this similarity can be measured, and the difference between two solutions cannot be arbitrarily large, since it is bounded.

Consider the value of the combined solution $u_{\mathbf{n}}^{(c)}$ at an arbitrary grid point $x_{\mathbf{l},\mathbf{j}}$ of the combined grid $\Omega_{\mathbf{n}}^{(c)}$. This value, $u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})$, is obtained from the combination of the different solutions $u_{\mathbf{i}}$ that include that grid point (with the appropriate combination coefficients). For every grid point $x_{\mathbf{l},\mathbf{j}}$ there is always at least one combination solution $u_{\mathbf{i}}$ that includes it, and at most $|\mathscr{I}|$ such grids. For example, all combination grids $u_{\mathbf{i}}$ include the grid points with $\mathbf{l} = \mathbf{1}$ ($x_{\mathbf{1},\mathbf{j}}$, corresponding to subspace $W_{\mathbf{1}}$). In other words, we have $|\mathscr{I}|$ solutions of the PDE at the grid points $x_{\mathbf{1},\mathbf{j}}$. Let's call $N_{\mathbf{l}} = 1, \ldots, |\mathscr{I}|$ the number of combination solutions $u_{\mathbf{i}}$ that contain the grid points $x_{\mathbf{l},\mathbf{j}}$. We expect the different versions of a point $u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})$ to be similar, but with slight variations. This variance is given by

$$\mathrm{Var}[u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] = \frac{1}{N_{\mathbf{l}}} \sum_{\mathbf{l}' \geq \mathbf{l}} \left(u_{\mathbf{l}'}(x_{\mathbf{l},\mathbf{j}}) - \mathrm{E}[u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})]\right)^2, \quad \mathbf{l}, \mathbf{l}' \in \mathscr{I}\ , \tag{30}$$

since a grid point $x_{\mathbf{l},\mathbf{j}}$ can be found in all combination solutions $u_{\mathbf{l}'}$ with $\mathbf{l}' \geq \mathbf{l}$ (see Eq. (9)). The mean value of $u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})$ over all combination solutions is defined as

$$\mathrm{E}[u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] = \frac{1}{N_{\mathbf{l}}} \sum_{\mathbf{l}' \geq \mathbf{l}} u_{\mathbf{l}'}(x_{\mathbf{l},\mathbf{j}})\ . \tag{31}$$

Since we have been working in the hierarchical basis, the variance of the value at point $x_{\mathbf{l},\mathbf{j}}$ in this basis is given by

$$\mathrm{Var}[\alpha_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] = \frac{1}{N_{\mathbf{l}}} \sum_{\mathbf{l}' \geq \mathbf{l}} \left(\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{l}')} - \mathrm{E}[\alpha_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})]\right)^2\ . \tag{32}$$

This quantity is in fact bounded, due to (26), by

$$\begin{aligned} \mathrm{Var}[\alpha_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] &= \frac{1}{2N_{\mathbf{l}}^2} \sum_{\mathbf{s} \geq \mathbf{l}} \sum_{\mathbf{t} \geq \mathbf{l}} \left(\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}\right)^2 \\ &\leq \frac{8 \cdot \kappa^2(x_{\mathbf{l},\mathbf{j}})}{N_{\mathbf{l}}^2} \sum_{\mathbf{s} \geq \mathbf{l}} \sum_{\substack{\mathbf{t} \geq \mathbf{l} \\ \mathbf{t} \neq \mathbf{s}}} g^2(h_{\mathbf{s}}^p, h_{\mathbf{t}}^p)\ , \end{aligned} \tag{33}$$

with $g(h_{\mathbf{s}}^p, h_{\mathbf{t}}^p) := h_{t_1}^p + h_{s_1}^p + h_{t_2}^p + h_{s_2}^p + h_{t_1}^p h_{t_2}^p + h_{s_1}^p h_{s_2}^p$.

Equation (33) tells us that if we observe how the solution of our PDE at point $x_{\mathbf{l},\mathbf{j}}$ varies among the different combination solutions, the variance will not be arbitrarily large. This gives us a second way to perform a sanity check to filter SDC, summarized in Algorithm 5. Using the fact that the variance of each point is bounded, we can apply existing algorithms from robust statistics to find outliers among the different versions of each point. The algorithms used are described in Sect. 4.1. This allows us to filter solutions with unusually large variation and we can be certain that the rest of the solutions has been computed correctly. Just as we did in the first implementation of `sdcSanityCheck`, we do not need to find a value for the upper bound of the variance (33), but simply to detect unusually large variations.

There is one special case to consider. What happens for subspaces $W_{\mathbf{l}}$ for which we only have one version of the solution ($N_{\mathbf{l}} = 1$)? These are the grid points found in the highest hierarchical subspaces (largest $\mathbf{l}$). In the very unfortunate case where one of these values is wrong *and the fault does not propagate to neighboring points*, we have no other values with which to compare it and thus it cannot be filtered with this approach (nor with the previous). A possible way to detect such errors can be deduced from the fact that the hierarchical coefficients should decrease exponentially in magnitude with increasing level $\mathbf{l}$ (Eq. (27)). This means that the coefficients on the highest hierarchical subspace should be very small compared to the rest. We verified this exponential decay for our advection problem as well as for the more complex plasma simulation code *GENE* [27]. One can try to verify that the hierarchical coefficients at the highest level are smaller than those at a lower level, say, m levels lower,

$$|\alpha^{(\mathbf{l})}_{\mathbf{l},\mathbf{j}}| < |\alpha^{(\mathbf{l})}_{\mathbf{l}-m\cdot\mathbf{e}_k,\mathbf{j}}| \,. \tag{34}$$

The direction $\mathbf{e}_k$ should be chosen preferably to be the most finely discretized one (i.e. that for which l_k is largest, which will be large in exascale simulations). For our experiments, $m = 3$ worked well. This check could return false positives if lower coefficients are small, so more robust checks could be useful. Note, however, that this check is not even necessary for the combination solutions with the finest discretization, since the combination step would not propagate the fault to other combination solutions. This further reduces the significance of this special case.

Algorithm 5 Sanity check via outlier detection

input: The set of all combination solutions $\{u_{\mathbf{i}}\}$ (in the hierarchical basis)
output: The set of indices corresponding to the solutions affected by SDC, $\{\mathbf{i}_{\text{sdc}}\}$
function SDCSANITYCHECK($\{u_{\mathbf{i}}\}$)
  for all grid points $x_{\mathbf{l},\mathbf{j}}$ in $\Omega^{(c)}_{\mathbf{n}}$ **do in parallel**
    $\alpha[\mathbf{l}'] \leftarrow$ GATHER($\alpha^{(\mathbf{l}')}_{\mathbf{l},\mathbf{j}}$) for all $\mathbf{l}' \geq \mathbf{l}$
    if any OUTLIER_TEST($\alpha[\mathbf{l}']$) **then**
      Add outlier $\mathbf{l}'$ to set of corrupted indices $\{\mathbf{i}_{\text{sdc}}\}$
  return $\{\mathbf{i}_{\text{sdc}}\}$

4 Numerical Tests

4.1 Experimental Setup

We implemented Algorithm 3 with both types of sanity checks (Algorithms 4 and 5) in Python for our 2D advection equation (15) with $c_x = c_y = 1$. Despite this being a toy problem, the algorithms presented are general enough to be applied to any PDE solver for which the SGCT converges. In other words, if a PDE can be solved using the SGCT, our sanity checks are guaranteed to work, thanks to the error splitting assumption (20). The function `solver` is a Lax-Wendroff solver, which has order two in space and time. For the FTCT we use a maximum resolution $\mathbf{n} = (9, 9)$ and $\tau = 3$ (giving $\mathbf{i}_{\min} = (6, 6)$). This results in a classical index set $\mathscr{I}_{\mathbf{n},q,\tau}$ with 7 elements and an extended set $\mathscr{I}^{\text{ext}}_{\mathbf{n},q,\tau}$ with 10 elements. We calculate the solution at time $t = 0.5$ using 512 total time steps for all combination solutions, which ensures that the CFL condition is met. We combine the solutions twice during the simulation (`reduce` step), once at the middle (after 256 time steps) and at the end (after 512 time steps).

For this discussion we use only the second version of the function `sdcSanity-Check` (detecting outliers), since we found it to be more robust and to have more potential for parallelization. In particular, the `gather` step can be combined with the `reduce` step, so we would only need to communicate once instead of twice. And second, this `gather+reduce` step can be performed efficiently using the algorithm *Subspace Reduce* [26] with small modifications.

All simulations presented here were carried out serially. To detect if any of the hierarchical coefficients $\alpha^{(\mathbf{l}')}_{\mathbf{l},\mathbf{j}}$ is an outlier, we used the Python library *statsmodels* [32], specifically, the function `outlier_test` from the module `linear_model`. As of version 0.7.0 the function implements seven outlier detection methods, all of which performed very similarly. For our tests we chose `method='fdr_by'` which is based on a false discovery rate (FDR) method described in [4]. We consider a grid to be affected by SDC if at least one of its values is detected as an outlier.

4.2 Results

In Fig. 5 we have plotted six sets of simulation results. Each set has an iteration number (from 0 to 511) on the x-axis, which represents the time step in the function `solver` at which SDC was injected. This means that each of the six plots shows 512 different simulations. On the y-axis we have plotted the L_2 relative error of the combined solution $u^{(c)}_{\mathbf{n}}$ with respect to the exact solution at the end of each simulation. The three rows of plots show the different magnitudes of the SDC (10^5, $10^{-0.5}$, and 10^{-300}). For all simulations, the wrong value was injected into the same combination solution, $u_{(7,8)}$. (Choosing different solutions made no difference in the

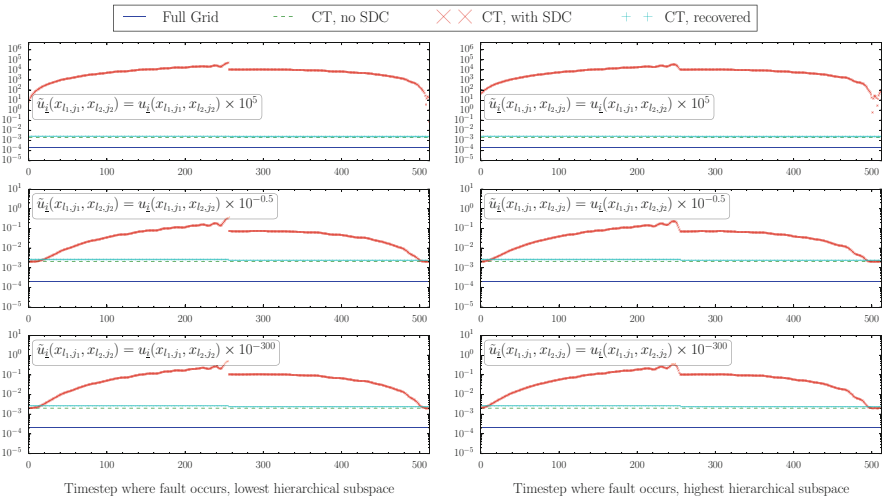

Fig. 5 L_2 relative error of the FTCT when SDC of various magnitudes are injected into one combination solution

results.) Finally, the plots on the left differ from those on the right by the choice of the grid point where SDC was injected. Recall that each combination solution $u_{\mathbf{i}}$ is transformed to the hierarchical basis after the call to the solver (line 10). If the solver returns a $u_{\mathbf{i}}$ with wrong values, the hierarchization step can propagate these wrong values to various degrees depending on which grid point(s) were affected. For the three plots on the left we injected SDC on the lowest hierarchical level (more precisely, in the middle of the domain), whereas for the plots on the right, the wrong value was inserted on one point of the highest hierarchical level (right next to the middle of the domain).

The blue line on each plot is the error of the full grid solution $u_{(9,9)}$; the green dotted line is the error of the combined solution $u_{\mathbf{n}}^{(c)}$ when no SDC is injected; each red dot represents the error of the combined solution when SDC has been injected and not filtered; and the blue crosses are the error of the combined solution after detecting and filtering the wrong solution ($u_{(7,8)}$) and combining the rest of the grids with different coefficients.

As discussed earlier, the error of the combined solution is proportional to the error of the wrong combination solution. In all but one of the $6 \times 512 = 3072$ simulations the wrong solution was detected and filtered. This was the case when SDC of magnitude $10^{-0.5}$ was injected on the lowest hierarchical subspace during the last iteration, because the value of the solution of the PDE at that point is very close to zero. (Recall that the exact solution is a product of sine functions, so it is equal to zero at various points.) This is also true during (roughly) the first and last ten iterations, and from the plots we can see that the outlier detection method as we applied it is *too* sensitive. Even when SDC is barely noticeable (thus not affecting the quality of the combined solution), it is still detected, and the recovered

solution (blue crosses) can actually be slightly *worse* than the solution with SDC (red circles). This is actually not too bad, since the error of the recovered combined solution is always very close to that of the unaffected combined solution, and we consider it a very unlikely worst-case scenario. This no longer happens if the value affected by SDC is not originally very close to zero. Some fine-tuning can be done to make the outlier detection method less sensitive. (Outlier detection functions usually involve a sensitivity parameter that can be varied.)

Whether SDC affects a point on the lowest hierarchical subspace or the highest makes almost no difference, but this is problem-dependent. In a different experiment, we added a constant to the initial field so that the solution is nowhere close to zero. This resulted in a higher error when SDC was injected in a low hierarchical subspace. One can also see that faults occurring in early iterations result in a larger error at the end. (Notice the small step in the blue crosses at iteration 256.)

There were some simulation scenarios where the wrong combination solution was not properly filtered, or when correct combination solutions were wrongly filtered. This happened when the minimum resolution of the SGCT ($\mathbf{i}_{\min}$) was too small. For our problem, the choice $\mathbf{i}_{\min} \geq (5, 5)$ was large enough for the outlier detection algorithms to work properly. As long $\mathbf{i}_{\min}$ is chosen large enough, both the SGCT and the sanity checks work as expected.

Finally, if SDC causes alterations in the control flow of the program, two scenarios are possible (assuming once again that the fault occurs when calling the solver): either the solver returns a wrong solution or does not return at all. The first case can be treated by our algorithms. The second case can either translate into a hard fault (i.e., an error signal is produced—and this can be dealt with) or cause the solver to hang indefinitely. We plan to investigate this last scenario in the future.

Despite these fine-tuning issues, our approach offers several advantages over existing techniques. We do not implement any complicated checksum schemes; there is no checkpointing involved at any memory level; and there is no need to replicate MPI processes nor data. We simply make use of the existing redundancy in the SGCT to either calculate a norm or to look for outliers. These two algorithms are inexpensive and should not be difficult to implement in parallel. We plan to investigate robust, parallel implementations in future work, as well as to carry out further experiments.

5 Conclusions

The SGCT and its fault tolerant version, the FTCT, offer an inherent type of data redundancy that can be exploited to detect SDC. Assuming that one or more combination solutions can be affected by SDC of arbitrary magnitude, one can perform sanity checks before combining the results. The sanity checks work even in the worst-case scenario where only one value in one field is affected by a factor of arbitrary magnitude. These recovery algorithms do not require checkpointing. Existing outlier detection techniques from robust statistics can be

directly incorporated into the FTCT, which requires only minimal modifications. Only some fine-tuning is required to minimize false positives or negatives, but this algorithmic approach avoids the drawbacks of the alternative techniques. We plan to demonstrate the applicability of these algorithms in massively large parallel simulations in the near future.

Acknowledgements This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA). We thank the reviewers for their valuable comments. A. Parra Hinojosa thanks the TUM Graduate School for financing his stay at ANU Canberra, and acknowledges the additional support of CONACYT, Mexico.

References

1. Ali, M.M., Strazdins, P.E., Harding, B., Hegland, M., Larson, J.W.: A fault-tolerant gyrokinetic plasma application using the sparse grid combination technique. In: Proceedings of the 2015 International Conference on High Performance Computing & Simulation (HPCS 2015), pp. 499–507. IEEE, Amsterdam (2015)
2. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
3. Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Ohlberger, M., Sander, O.: A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. Computing **82**(2–3), 103–119 (2008)
4. Benjamini, Y., Yekutieli, D.: The control of the false discovery rate in multiple testing under dependency. Ann. Stat. **29**(4), 1165–1188 (2001)
5. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Post-failure recovery of MPI communication capability: design and rationale. Int. J. High Perform. Comput. Appl. **27**(3), 244–254 (2013)
6. Bridges, P.G., Ferreira, K.B., Heroux, M.A., Hoemmen, M.: Fault-tolerant linear solvers via selective reliability. Preprint arXiv:1206.1390 (2012)
7. Bungartz, H.J., Griebel, M.: Sparse grids. Acta Numer. **13**, 147–269 (2004)
8. Chen, Z., Dongarra, J.: Highly scalable self-healing algorithms for high performance scientific computing. IEEE Trans. Comput. **58**(11), 1512–1524 (2009)
9. van Dam, H.J.J., Vishnu, A., De Jong, W.A.: A case for soft error detection and correction in computational chemistry. J. Chem. Theory Comput. **9**(9), 3995–4005 (2013)
10. Elliott, J., Hoemmen, M., Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1193–1202. IEEE (2014)
11. Elliott, J., Hoemmen, M., Mueller, F.: Resilience in numerical methods: a position on fault models and methodologies. Preprint arXiv:1401.3013 (2014)
12. Ferreira, K., Stearley, J., Laros III, J.H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 44. ACM (2011)
13. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 78. IEEE Computer Society Press (2012)

14. Garcke, J.: A dimension adaptive sparse grid combination technique for machine learning. ANZIAM J. **48**, 725–740 (2007)
15. Garcke, J.: Sparse grids in a nutshell. In: Garcke, J., Griebel, M. (eds.) Sparse Grids and Applications. Lecture Notes in Computational Science and Engineering, pp. 57–80. Springer, Berlin/Heidelberg (2013)
16. Garcke, J., Griebel, M.: On the computation of the eigenproblems of hydrogen and helium in strong magnetic and electric fields with the sparse grid combination technique. J. Comput. Phys. **165**(2), 694–716 (2000)
17. Griebel, M.: The combination technique for the sparse grid solution of PDE's on multiprocessor machines. Parallel Process. Lett. **2**, 61–70 (1992)
18. Griebel, M., Schneider, M., Zenger, C.: A combination technique for the solution of sparse grid problems. In: Iterative Methods in Linear Algebra, pp. 263–281. IMACS, Elsevier, North Holland (1992)
19. Harding, B.: Adaptive sparse grids and extrapolation techniques. In: Sparse Grids and Applications. Lecture Notes in Computational Science and Engineering, pp. 79–102. Springer, Cham (2015)
20. Harding, B., Hegland, M., Larson, J., Southern, J.: Fault tolerant computation with the sparse grid combination technique. SIAM J. Sci. Comput. 37(3), C331–C353 (2015)
21. Heene, M., Kowitz, C., Pflüger, D.: Load balancing for massively parallel computations with the sparse grid combination technique. In: PARCO, pp. 574–583. IOS Press, Garching (2013)
22. Heene, M., Pflüger, D.: Scalable algorithms for the solution of higher-dimensional PDEs. In: Proceedings of the SPPEXA Symposium. Lecture Notes in Computational Science and Engineering. Springer, Garching (2016)
23. Heene, M., Pflüger, D.: Efficient and scalable distributed-memory hierarchization algorithms for the sparse grid combination technique. In: Parallel Computing: On the Road to Exascale, Advances in Parallel Computing, vol. 27, pp. 339–348. IOS Press, Garching (2016)
24. Hegland, M.: Adaptive sparse grids. ANZIAM J. **44**, C335–C353 (2003)
25. Hupp, P.: Performance of unidirectional hierarchization for component grids virtually maximized. Procedia Comput. Sci. **29**, 2272–2283 (2014)
26. Hupp, P., Jacob, R., Heene, M., Pflüger, D., Hegland, M.: Global communication schemes for the sparse grid combination technique. Adv. Parallel Comput. **25**, 564–573 (2013). IOS Press
27. Jenko, F., Dorland, W., Kotschenreuther, M., Rogers, B.N.: Electron temperature gradient driven turbulence. Phys. Plasmas **7**(5), 1904–1910 (2000). http://www.genecode.org/
28. Kowitz, C., Hegland, M.: The sparse grid combination technique for computing eigenvalues in linear gyrokinetics. Procedia Comput. Sci. **18**, 449–458 (2013)
29. Parra Hinojosa, A., Kowitz, C., Heene, M., Pflüger, D., Bungartz, H.J.: Towards a fault-tolerant, scalable implementation of gene. In: Recent Trends in Computational Engineering – CE2014. Lecture Notes in Computational Science and Engineering, vol. 105, pp. 47–65. Springer, Cham (2015)
30. Pflüger, D.: Spatially Adaptive Sparse Grids for High-Dimensional Problems. Verlag Dr. Hut, München (2010)
31. Reisinger, C., Wittum, G.: Efficient hierarchical approximation of high-dimensional option pricing problems. SIAM J. Sci. Comput. **29**(1), 440–458 (2007)
32. Seabold, S., Perktold, J.: Statsmodels: econometric and statistical modeling with python. In: Proceedings of the 9th Python in Science Conference, pp. 57–61 (2010). http://statsmodels.sourceforge.net/
33. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., et al.: Addressing failures in exascale computing. Int. J. High Perform. Comput. Appl. **28**, 129–173 (2014)
34. Winter, H.: Numerical advection schemes in two dimensions (2011). www.lancs.ac.uk/~winterh/advectionCS.pdf