Technische Universität München
Fakultät für Informatik
Lehrstuhl für Wissenschaftliches Rechnen (I5, Prof. Bungartz)

Toward Resilient Exascale PDE Solvers Using the Combination Technique

Alfredo Parra Hinojosa

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des Akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende(r): Prof. Dr. Hans Michael Gerndt

Prüfer der Dissertation: 1. Prof. Dr. Hans-Joachim Bungartz

2. Jun.-Prof. Dr. rer. nat. Dirk Pflüger

Die Dissertation wurde am 22.09.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.11.2017 angenommen.

Abstract

As supercomputers become larger and more complex, application developers face new challenges. Massively parallel systems not only allow scientists to enhance their codes and achieve unprecedented levels of performance – system *faults* are no longer invisible to developers, which means they also have to *protect* their applications from unexpected (and unwanted) behavior. The interplay between maximizing performance while ensuring simulations run successfully on increasingly unreliable hardware is the topic of much research in high-performance computing, and the focus of this thesis.

In particular, we study a class of problems that have historically benefited from advances in HPC and now therefore face the challenges brought about by system faults. We want to solve *partial differential equations* (PDEs), which play a central role in the simulation sciences. In particular, we look at PDEs defined on higher-dimensional domains (more than three dimensions) which, in realistic simulation scenarios, often require vast computing resources. We present a prototypical example of such an application: gyrokinetic plasma simulations used by physicists to better understand anomalous and turbulent plasma flow in fusion reactors, the main limiting factor in the path to clean fusion energy.

We describe an algorithm that can exploit the computing resources of massively-parallel systems to solve such PDEs: the *sparse grid combination technique* algorithm. Thanks to the work of our partners at the *IPVS Stuttgart*, this algorithm can be run efficiently on high-end computing systems. Our goal for this thesis is to make sure that the algorithm runs successfully despite any hardware or software errors, but without sacrificing the code's performance. To do this, we make use of both numerical mathematics and software engineering, and we argue that understanding the mathematical properties of an algorithm is the key to develop efficient resilient codes.

This work is a building block of the project *EXAHD*, which is part of the German Priority Programme *Software for Exascale Computing* (SPPEXA). *EXAHD* is a joint collaboration between the Technical University of Munich, the University of Stuttgart, the University of Bonn and the Max Planck Institute for Plasma Physics.

Acknowledgements

First and foremost, I'd like to thank Hans Bungartz for acting as my supervisor throughout my doctoral work, for offering his support whenever I requested it, and for trusting me with a wide range of responsibilities during my time at TUM. His work ethic is inspiring and contagious.

I'm also deeply thankful to Dirk Pflger for his constant support, as well as his valuable and detailed feedback on every one of my publications and talks. His attention to detail consistently improved the quality of the work we did at EXAHD – he is the main driving force behind the success of the project.

I am grateful I had the chance to work closely with Mario Heene of the University of Stuttgart. It was not only a fruitful and productive collaboration but also always friendly and fun.

I'm especially thankful to Christoph Kowitz, Michael Obersteiner, Brendan Harding, and Markus Hegland for the invaluable scientific collaboration.

My colleagues at the Chair of Scientific Computing were an incredible source of support and friendship, especially Christoph Riesinger, Tobias Neckel, and Michael Rippl as CSE coordinators. I'm also indebted to Emily Mo-Hellenbrand, Ionut-Gabriel Farcas, Paul Sarbu, Michael Obersteiner, and Mario Heene for revising parts of my thesis.

The EXAHD project is possible thanks to the German Research Foundation's (DFG) initiative *SPPEXA: Software for Exascale Computing.* For additional financial support, I'd like to thank Conacyt, Mexico.

Finally, I thank my parents and sisters for their unconditional love and support.

Contents

1 Introduction

A wide range of interesting physical phenomena can be modeled using partial differential equations (PDEs). Since this class of problems will be the focus of our thesis, it is worthwhile considering some of the challenges that PDEs present to high-performance computing (HPC). As a motivating example, consider the problem of simulating hot plasma confined by a strong magnetic field. High-resolution simulations of confined plasma are of great value to physicists and engineers building efficient fusion power plants, since plasma fusion is considered one of the most promising alternatives to generate clean, safe and sustainable energy for future generations [KGHW15]. But developing this technology is as challenging as it is exciting. The principal obstacle standing on the way of efficient plasma power plants is the anomalous transport phenomena that arise at such high temperatures (typically reaching the order of magnitude of 100 million Kelvin) [GLB$^+$11]. This behavior is mainly caused by small-scale turbulence, which has been an elusive problem in plasma physics for a long time.

The evolution of a plasma field in this state can be described by a PDE (actually, a system of PDEs), namely, the *gyrokinetic Vlasov-Maxwell equations.* Although the system has a complicated form, it can has the following general form [KPJH12]

$$\frac{\partial u}{\partial t} = \mathcal{L}(u) + \mathcal{N}(u)\,. \tag{1.1}$$

This system characterizes the time evolution of a (5+1)-dimensional plasma field whose particles are described by a probability distribution $u \equiv u(x, y, z, v_{\|}, \mu; t)$. This distribution depends on the spatial coordinates x, y and z as well as on the two velocity coordinates $v_{\|}$ and μ. The differential operators $\mathcal{L}$ and $\mathcal{N}$ describe the linear and nonlinear evolution of u in the five-dimensional domain, respectively.

Solving Eq. (1.1) is a challenging task. The system is not only nonlinear but also coupled, and both $\mathcal{L}$ and $\mathcal{N}$ are integro-differential operators. One physics code that attempts to solve this system numerically is `GENE` [J$^+$00]. This code uses a Runge-Kutta scheme in time and a combination of high-order finite differences and Fourier discretization in space. The resulting five-dimensional Cartesian grid $\Omega_{\mathbf{i}}$ has $2^{i_1} \times 2^{i_3} \times 2^{i_3} \times 2^{i_4} \times 2^{i_5}$ discretization points. Additionally, the code can simulate the interaction of various types of particles (called *species*), such as electrons and ions, and each particle has its own 5D grid.

A report by the developers of `GENE` for the Jülich Supercomputer center describes some typical simulation scenarios and the respective memory costs [MF10]:

- **Multi-scale problems**, which investigate the coupling of electrons and ions, require grid sizes with typical sizes of $1024 \times 512 \times 24 \times 48 \times 16$ for each species ($= 2^{33}$ in total). The memory requirements for this type of problems are of the order of 2.5 TB.
- **Stellerator problems** attempt to model the more complicated geometry of stellerator fusion reactors. A typical simulation scenario with two species has $128 \times 64 \times 512 \times 64 \times 16$ grid points per species ($\approx 2^{34}$ in total) and a memory footprint of 2 TB.
- **Global simulations** extend the computational domain along the radial dimension of the fusion reactor, which requires considerably more points this direction. Many scenarios of interest in this simulation mode are currently not feasible, but projected grids have $8192 \times 64 \times 32 \times 128 \times 64$ grid points and four species ($= 2^{39}$ grid points total). Such simulations will require roughly 100 TB of memory.

These huge computational requirements pose a big challenge for the plasma physics community, slowing down the progress in plasma fusion research. Scientists can either wait for supercomputers to offer the resources needed to run such simulations, or apply novel techniques from the field of numerical mathematics to optimize their codes now.

The memory requirements, as challenging as they might be, are actually only one of many problems that computational scientists face in light of the approaching *exascale* era. Scientists expect to have computing systems capable of performing at least 10^{18} (a quintillion) floating point operations per second in the time frame of 2018–2020 [DBM+11]. Numerical computations at this scale will require unprecedented resources in terms of core-hours and the issues arising in current petascale systems will be magnified in exascale computers. One such problem will be the main topic of this thesis, namely, the frequent occurrence of *errors* which cause computing systems to fail or return erroneous results.

Computing systems have always experienced different types of errors, and although current architectures are extremely robust to anomalous behaviors, the mere scale of the number of components that exascale computers will be built of means that errors will be inescapable. Furthermore, they will occur *often*, as we will see in Chapter 2.1 when we look at probabilistic estimates of errors in exascale.

At this point we can state the main goal of this thesis, as motivated by our discussion so far. Our aim is *to apply and extend state-of-the-art numerical algorithms that can reliably run on future exascale systems prone to errors of various kinds in order to solve high-dimensional PDEs.* This might sound like a daunting task, and we do not claim to have a one-size-fits-solution to every high-dimensional PDE solver. What we instead hope to achieve in this thesis is to show how *we* moved from having existing, peta-scalable PDE solvers that are vulnerable to system faults, to robust codes that can be reliably run on error-prone supercomputers. Our hope is that our ideas can inspire similar efforts with different types of simulation codes – not just solvers for high-dimensional PDEs. In this spirit we will share not only the ideas that worked best but also those that were not as successful and the lessons learned. That being said, we can dive right into the topic of fault-tolerant PDE solvers.

2 Fault Tolerance in High-Performance Computing

Since petascale performance was achieved back in 2009, node and core count have kept increasing in HPC systems. This has opened the doors to many new exciting applications, but the increasing complexity of petascale computers has also given rise to complex challenges at almost all system levels. With some scientists expecting the first exascale system to appear as early as 2020, it seems reasonable to start addressing some of the problems that will most likely affect these systems. Some of these problems are a result of extrapolating the behavior of existing petascale machines, while others might be particular to this new computing scale. For example, we know that the supercomputer *Blue Waters* had a *mean time between failures* (MTBF) of 4.2 hours during 2013 [DM$^+$14]. What can we conclude for systems that will have one or two orders of magnitude more components? Or what will be the energy requirements of a supercomputer with $\mathcal{O}(10^6)$ nodes, and how will that be reflected on a code's performance?

In this chapter we want to discuss some of the most pressing issues that exascale systems will face. More specifically, we will talk about the challenge of fault tolerance, the role it plays in current systems and how it will most likely become a ubiquitous part of algorithm design in the near future. We will cover the different types of system faults and give an overview of the techniques used to overcome them and discuss whether current techniques are likely to be applicable in the future. We will finish by mentioning some further problems that might be particular to exascale systems. This will give us a rationale for the motivation behind our work.

2.1 Towards Exascale

HPC systems keep increasing in size at almost every level, from total node count to node memory and IO bandwidth. Table 2.1 summarizes some orders of magnitude of typical HPC systems since 2009, and what we can expect with the arrival of the first exascale computer. Although the specification details of exascale machines are still largely unknown (for example, whether they will be equipped with $\mathcal{O}(10^5)$ nodes with $\mathcal{O}(10^4)$ cores each – the *fat node* scenario – or with $\mathcal{O}(10^6)$ nodes with $\mathcal{O}(10^3)$ cores each – the *slim node* scenario [DHR15]), one can already predict

Systems	**2009**	**2011**	**2015**	**2018 (??)**
System peak	2 Peta	20 Peta	100-200 Peta	1 Exa
System memory	0.3 PB	1.6 PB	5 PB	10 PB
Node performance	125 Giga	200 Giga	200-400 Giga	1-10 TF
Node memory BW	25 GB/s	40 GB/s	100 GB/s	200-400 GB/s
Node concurrency	12	32	$\mathcal{O}(10^2)$	$\mathcal{O}(10^3)$
Interconnect BW	1.5 GB/s	22 GB/s	25 GB/s	50 GB/s
# nodes	18,700	$100,000$	$500,000$	$\mathcal{O}(10^6)$
Total concurrency	225,000	3,200,000	$\mathcal{O}(50,000,000)$	$\mathcal{O}(10^9)$
Storage	15 PB	30 PB	150 PB	300 PB
IO	0.2 TB/s	2 TB/s	10 TB/s	20 TB/s
MTTI	4 days	19 h 4 min	3 h 52 min	1 h 56 min
Power	6 MW	10 MW	10 MW	20 MW

Table 2.1: Roadmap towards exascale computing (by C. Engelmann and S. Scott, as presented in Y. Robert's fault tolerance tutorial at Euro-Par 2016 [Rob16]). The figure 2018 seems too optimistic. As of this writing, exascale computers will most likely arrive in 2020 at the earliest.

some of the problems expected at such a scale.

The authors of the *International Exascale Software Project* [DB+11] identify five parameters that will be critical when approaching exascale:

1. *Concurrency*: Considering Moore's law and Dennard scaling, exascale applications may run on up to ten billion threads.

2. *Power Consumption*: The high power consumption of exascale machines (possibly over 100 MW) will likely shift the focus from *operations per second* to *energy to solution* metrics.

3. *Resiliency*: Future HPC systems will be made up of increasingly *unreliable* components, and applications need to be aware of this.

4. *Heterogeneity*: Both at the hardware level (CPUs, GPUs, etc.) and at a software level (multi-scale simulations).

5. *I/O and memory*: There will likely be new memory hierarchies, which will affect current programming models.

We will focus on the third problem: what should we consider when running applications on systems that are increasingly prone to faults?

In order to address this question, we first need to define a few concepts that appear often when discussing resiliency. The concept of *resiliency* itself refers to "the techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults" [C+14]. A closely related term is *fault tolerance*, which is achieved by "detecting errors and notifying about errors and by recovering, or compensating for errors" [C+14]. These two terms are often used interchangeably, and we will also do so throughout this thesis.

There are also different things that can go wrong in an HPC system. It has become customary to use the terms introduced by Avižienis et al. [ALRL04] when talking about resilience. These terms have also been summarized in [SWA+14] (Section 2, *Taxonomy of terms*). The terms we will encounter most often include

- *Failure*: Deviating from the correct service of a system function.
- *Error*: The part of the system state that may lead to a failure (such as a bad value).
- *Fault*: The cause of an error (such as software bugs or alpha particles hitting the dye).

Let us take a look at some of the main characteristics of failures, errors and faults:

- *Failure*
  - Domain: What failed? (Incorrect state, wrong timing, ...)
  - Persistence: Does the system halt completely or does it exhibit erratic behavior?
  - Detectability: Did the failure produce a signal to the user? Is the signal correct?
  - Consistency: Do all users receive the same signal? (If not, the failure is called *byzantine*.)
- *Error*
  - Detected: It generated a signal.
  - Latent/Silent: Not detected.
  - Masked: Does not cause a failure.
- *Fault*
  - Active: Causes an error.
  - Dormant: Does not cause an error.
  - Permanent: Fault persists over time.
  - Transient: Fault appears temporarily.
  - Hard: Cause can be systematically traced back and reproduced.
  - Soft: Cause cannot be systematically traced back and reproduced.

The relative importance of the various types of faults and errors depends on different factors, including the size of the system, the stage at which the system is being operated[1] and the voltage supply, among others [SWA+14]. But given that nowadays *at least 20% of computing resources in HPC systems is wasted as a cause of failures* [EBEG+08] it should be clear that this is an urgent problem to address. Experts agree that faults in high-end HPC systems are the norm rather than the exception [FME+12]. The Tsubame 2 supercomputer in Tokyo suffered 962 faults in 18 months (mean time between failures, or MTBF, is thus 13 hours); the Blue Waters system at the University of Illinois at Urbana-Champaign reported 2-3 node failures per day; and the petascale system Titan at the Oak Ridge National Laboratory has also been reported to fail several times per day [Rob16]. Even if an individual node has an MTBF of, say, 100 years, a system with 100k such nodes is bound to fail every 9 hours on average [DHR15]. In exascale, where we could have systems with several million processors, jobs could fail as frequently as every 30 minutes [SWA+14].

These conclusions are the result of a simple theorem [Rob16]:

If the MTBF of one processor is μ, then the MTBF of p processors is $\mu_p = \frac{\mu}{p}$.

This holds for any distribution of faults.

It should be noted that a large percentage of the errors occurring in HPC systems are caused by the software (application bugs, OS malfunctioning, errors in the file system, etc.), but hardware errors are much more expensive to correct. In one 2005 study, software errors accounted for 59% to 84% of all outages, but they need 0.6 to 1.5 hours to repair, whereas hardware errors sometimes require up to 100 hours [CD05]. A more recent study of the Blue Waters petascale machine, however, concluded that software failures[2] represented only 20% of the total failures over a lapse of 261 days, but contributed 53% of the total repair time [DM+14]. Their study indicated that hardware is very resilient to faults, with error correcting codes detecting and fixing 99.997% of all errors, which is notable given that the system experienced system-wide outages every 159 hours on average. Over the period studied, hardware was responsible for 51% of all single and multiple node failures. Unfortunately, given that exascale systems are expected to be composed of increasingly unreliable components, it is unlikely that this high level of resiliency will be sustained.

System software errors as described above have to be handled by systems experts. Given that our expertise is algorithm design and numerics, we are in a position to address mainly two type of errors: detected errors caused by hard faults and any unmasked silent errors. As we mentioned earlier, hard faults can be traced back and reproduced – these are mainly caused by internal errors, such

[1]Supercomputers are more prone to failures at the early operation stage (*infant mortality*) as well as near the end of their usable life. In between, failure rates remain largely constant.

[2]These exclude failures caused by the application software, and can be categorized into three types: 1) pure software errors (unhandled exceptions, incorrect return values, concurrency errors, overflows, etc.), 2) software not handling hardware errors (unhandled node failures or disk failures resulting in file system failures) and 3) software causing errors in the hardware (for example, due to incorrect firmware). Including application software failures would increase the percentage of total software failures even further [DM+14].

as device degradation or low voltage operation [SWA+14]. Silent errors, on the other hand, cannot be systematically reproduced, which is mainly due to the fact that they are caused by external factors, such as alpha particles hitting the dyes. Unmasked silent errors are usually referred to as *Silent Data Corruption* (SDC). These two types of errors will play a central role in this thesis, so let us say a few more words about them.

2.2 Hard Faults

Hard faults refer primarily to malfunctioning hardware, and as we already discussed, they may lead to system failures. In our work, we are interested in overcoming *fail-stop failures*, which cause running applications to interrupt their normal execution. In the previous section we presented a few numbers regarding the frequency of different types of faults in petascale systems. In order to come up with strategies to deal with these faults, it is useful to try to model the frequency at which they hit the system. Since the frequency is not deterministic, it is common to use probability distributions to model fail-stop failures. One common simplifying assumption is that the times between any two faults are independent and identically distributed (i.i.d.) random variables[3]. In other words, if X_i denotes the time elapsed between fault i and fault $i+1$, then every X_i is i.i.d. and its probability distribution can be approximated by an exponential distribution [DHR15],

$$f(t,\lambda) = \lambda e^{-\lambda t}. \tag{2.1}$$

This is convenient because exponential distributions are memory-less, which is expressed as

$$\mathbb{P}(T \geq t+s | T \geq s) = \mathbb{P}(T \geq t), \quad \forall t, s \geq 0. \tag{2.2}$$

We can also use this distribution to define the mean time between failures more precisely as $\mu = \mathbb{E}(X) = \frac{1}{\lambda}$. The parameter λ is then the instantaneous failure rate of the system.

However, it is more common to use the closely-related *Weibull distribution* for more realistic models, since this distribution can account for the higher failure rate at the early operation stage of a system (*infant mortality*). This distribution is given by

$$f(t,\lambda,k) = k\lambda(t\lambda)^{k-1}e^{-(\lambda t)^k}, \tag{2.3}$$

and the cumulative distribution function by

$$F(t,\lambda,k) = 1 - e^{-(\lambda t)^k}. \tag{2.4}$$

Figure 2.1(left) shows the shape of $F(t,\lambda,k)$ for $\lambda = 0.1$ and $k = 0.5, 0.7$, along with the corresponding exponential distribution ($k = 1$). This is the probability of failure of one processor. The parameter k is chosen such that the MTBF increases

[3]This is not true in general, since processors are more likely to fail if other processors nearby failed earlier. However, this approximation is still quite good for practical purposes [Rob16].

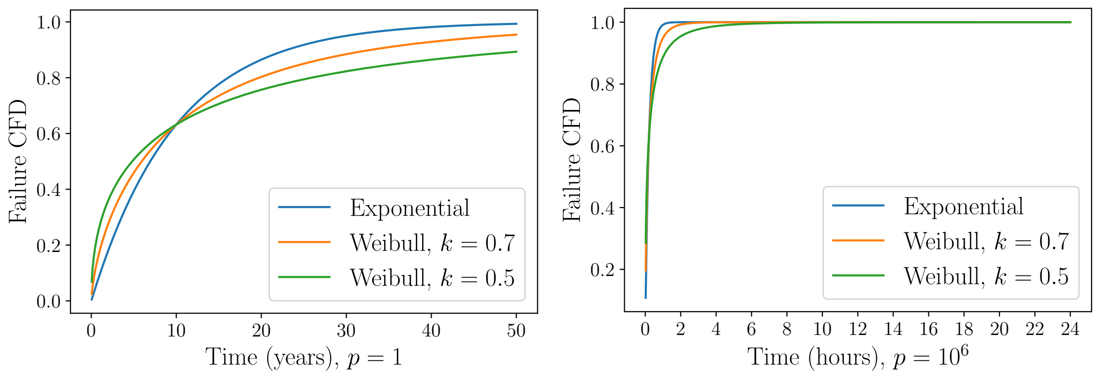


Figure 2.1: Exponential and Weibull cumulative distribution functions with $\lambda = 0.1$ for a machine with $p = 1$ processor (left) and $p = 10^6$ processors (right). The Weibull distribution models infant mortality better. The values $k = 0.5$ and $k = 0.7$ are common in the literature and model real system behavior quite accurately.

with time (faults become less frequent), which is the case for any $k < 1$. In this case, the MTBF is given by

$$\mu = \mathbb{E}(X) = \frac{1}{\lambda}\Gamma\left(1 + \frac{1}{k}\right).$$

As we saw in the previous section, one can substitute μ with μ/p to model the behavior of a parallel system. The resulting CFD with $p = 10^6$ can be seen on the figure on the right hand side.

By now, hard faults have been studied extensively and there is consensus within the HPC community about the importance of overcoming this problem in existing petascale and future exascale systems. There is a wide variety of approaches to make applications tolerant to faults, and since this is the main topic of this thesis, we will now take a look at the most common and briefly discuss their advantages and disadvantages. An excellent reference on this topic is the monograph compiled by Herault and Robert, on which the following discussion is based [DHR15].

2.2.1 Recovery Strategies

The most common and widely-used approach to deal with hard faults is *checkpointing*. Since hard faults usually cause data to go lost, one could try to store the state of an application to a memory level that is not affected by the faults. This state, or checkpoint, can be the entire data necessary to run the application at a given point, or only parts of it. If an application is affected by a fault, the state can be restored from the last successfully stored checkpoint instead of restarting the whole application. Checkpointing requires two steps: generating the checkpoint and storing it to a safe memory space. Generating the checkpoint can usually be overlapped with computation, and in some cases one can create duplicate processes for this stage, such that the duplicate generates the checkpoint and the parent continues execution. Once the checkpoint file is created, it is stored to memory, which can be physically close to the process or at a remote level, depending on the architecture and the risk of memory corruption. The main advantage of checkpointing techniques is that they can be general purpose: they can be implemented at a level where the application does not notice it.

There are different protocols to perform checkpointing and to restart applications after faults. They mainly differ in the choice of which processes do the checkpointing and at which points in time.

Process Checkpointing

In order to come up with sophisticated checkpointing protocols, it is important to determine how each single process will store its state at a given point. Users typically choose at what level they want to perform the checkpoint, which can range from very low level operating system calls, to high level, application-specific functions defined by the user at certain parts of the algorithms. Since a process can consist of several threads, a blocking call is usually necessary, and the process can create the checkpoint itself or create a duplicate that does it. It is then stored to the memory hierarchy defined by the user.

Coordinated Checkpointing

Having defined a protocol for a single process checkpoint, one needs to deal with distributed systems with many processes. The idea behind coordinated checkpointing is for all processes in a system to have the same consistent view of the total state of the application at a certain point. This is done by coupling process checkpointing with message passing to transmit information among processes. The main difficulty lies in determining what to do with messages sent among processes that are performing checkpoints. A consistent protocol should be able to keep track of the list of messages sent before and after every process executes a checkpoint, so that the exact sequence of messages can be reproduced during the recovery phase.

If a failure occurs, all processes restart from the last consistent state stored. The main disadvantage is that simply informing all processes that recovery has to be done becomes more expensive as the number of processes grow, so coordinated checkpointing can quickly run into scalability issues.

Uncoordinated Checkpointing

Instead of forcing all processes to make checkpoints simultaneously, one alternative would be for processes (or groups of processes) to make checkpoints independently of each other in order to reduce the communication overhead. Then, if faults occur, only the affected processes restart from their last checkpoint, but some additional information from other processes is necessary. Ensuring consistency becomes very problematic, since some operations might not be deterministic (such as the order in which MPI messages are received). This then requires message logging, which adds a large overhead to the protocols.

Hierarchical Checkpointing

Coordinated checkpointing has the advantage of being able to offer consistent views of the application's state at the cost of having to synchronize all processes in the system. Uncoordinated checkpointing avoids the synchronization overhead but introduces complex message logging in order to ensure consistency. Hierarchical checkpointing tries to find a balance between both by defining groups of processes that perform coordinated checkpointing and relying on message logging for all events involving different groups. This avoids global restarting.

In-Memory Checkpointing

Since one of the main costs of checkpointing is storing the data to stable memory, there have been efforts to avoid these expensive memory accesses and instead use the processes' main memory to store the checkpoints, see e.g. [ZNK12]. Of course, such an approach has the risk of failing under certain circumstances, but if fatal scenarios can be prevented or their risk minimized, this can be a promising and scalable alternative. One way to do this is to arrange processors into *buddy* pairs that duplicate and exchange checkpoints. Each process then has its own checkpoint as well as that of its buddy. Processes carry out this data exchange when they are not sending or receiving messages. This protocol can be combined with other protocols in order to avoid certain fatal scenarios.

* * *

All of the above checkpointing protocols can be supplemented with other strategies to make them more robust or scalable, such as trying to predict when faults will occur, replicating processes or using the algorithm's mathematical properties or data structures.

Fault Prediction

The most important question to answer when implementing a checkpointing strategy is how often to perform checkpoints. The checkpointing interval is usually calculated using information about the MTBF of the system, which is usually approximated since it is in general not known. If one could predict when faults will occur, one could take checkpoints at well-defined times and therefore less often than the models suggest. There are several ways to try to predict when a fault will occur. For example, some researchers have used Bayesian networks to learn from previous system faults and use this information to predict future faults [SOR+03]. Others have used genetic algorithms to even try to predict which parts of the system will be affected in order to avoid global checkpointing [ZLG+10]. It is important that prediction algorithms can predict as many faults as possible (called high *recall*), that the predictions made are correct (with high *precision*) and that the prediction gives the application enough time to perform the checkpoint (sufficient *lead time*).

As systems become larger and faults become more common, prediction algorithms might also become more robust since they can use more information about the state of the system when faults occur.

Replication

The idea if replication is straightforward: make replicas of every process so that if a given process is affected by faults, the replica can still continue computations. It might seem like many computational resources are wasted, since only half of the processes effectively do work. But if the process count is high and the usual checkpointing strategies require too many resources, replication can be profitable.

The idea is as follows. The probability of faults affecting two replicas simultaneously is quite low, but nonzero. One can compute the probability of this happening and deduce a checkpointing interval based on this probability. In other words, one performs checkpointing only if both replicas are likely to be affected. Evidently, these intervals are much longer than those computed without replicas, making replication a favorable approach when process count is high.

* * *

All the techniques we have described so far rely on what is usually referred to as *backward recovery*, which simply means that we try to trace back and reproduce the application's state at an earlier time and restart from there[4]. This involves repeating some operations but such approaches can often be used for general-purpose applications. However, it is not clear whether checkpointing alone will be enough in exascale, especially considering that the time required to write checkpoints could be of the same order of magnitude as the system's mean time to failure [C$^+$14].

Some of these drawbacks can often be overcome by understanding the details of a given application and trying to come up with fault tolerance mechanisms that are specific to each application. For example, if parts of the numerical data are not crucial for the algorithm's recovery, it would make little sense to store them to safe memory. Maybe the data can be reconstructed from other data, or maybe some loss in accuracy can be tolerated. One could then perform *forward recovery* and try to continue with the computations without rolling back to an earlier state. *Algorithm-Based Fault Tolerance* (ABFT) is a broad class of techniques that rely on forward recovery by analyzing and exploiting the numerical properties of the application, and they represent a promising approach to fault tolerance for high-end HPC systems.

[4]Replication alone does not involve backward recovery – only used together with a checkpointing strategy.

Algorithm-Based Fault Tolerance

The term ABFT was first coined in Huang and Abraham's 1987 paper *Algorithm-based fault tolerance for matrix operations* [HA84]. They were interested in carrying out matrix operations such as addition, multiplication, and LU decomposition reliably on a system with faults. Their method made use of the fact that certain matrix quantities are invariable to various operations, so any missing or wrong data could be deduced by adding some redundancy, usually *checksums* – sums over rows, columns or matrix blocks. Although the term ABFT is sometimes used synonymously with this kind of linear algebra algorithms (and Huang and Abraham's ideas are still being extended [BDDL09]), we will use it to refer to any approach that exploits the numerical properties of any algorithm in order to make it fault tolerant[5]. It turns out that many classes of algorithms have properties that make them resilient to faults, such as different iterative schemes [BFHH12,Che11]. Some researchers are even devising machine learning techniques to identify and design resilient algorithms based on their numerical stability [CSM14].

Addressing fault tolerance at the algorithmic level has several advantages, including [Rob16]:

- Smaller checkpoint sizes (or none at all)
- Portability (since usually hardware-independent)
- Higher flexibility in the choice of parameters

Evidently, ABFT can be combined with any of the backward recovery techniques described previously, the reasoning being: let the application recover to the extent allowed by its mathematical properties, and for all other cases use checkpointing (or any variation thereof). An example of such a hybrid approach for generic linear algebra routines is described in [BBH$^+$14].

The general sentiment of the fault tolerance community is summarized well in the famous report *Toward exascale resilience*, where the authors argue that "improvements of current methods will not be sufficient to meet the failure challenges present in exascale systems. [...] Having the applications and their algorithms indifferent to fault-errors-failure would be an ultimate goal of the resilience community." [C$^+$09]. Investigating new approaches to fault tolerance (beyond checkpointing) is the main goal of this thesis.

2.3 Silent Errors

The second most important category of errors – at least if we use the amount of research invested as a proxy for importance – are silent errors. When unmasked (i.e., when they cause a failure in the application), silent errors usually manifest themselves as arithmetic computation errors, control flow errors, or as data not properly transferred through the network [SWA$^+$14]. Unmasked silent errors –

[5]The term *Application-Specific Fault Tolerance* is also sometimes used.

also known as silent data corruption, or SDC – are still poorly understood. Some call them the "monster in the closet" [Gei11] since there is little data available about their frequency. But some argue that silent errors are also subject to the theorem $\mu_p = \mu/p$, so we should expect their frequency to increase with processor count. Furthermore, other have remarked that one single occurrence of SDC can have fatal consequences in a simulation [CPHM08, EHM14a].

2.3.1 Detection and Recovery Strategies

Compared to errors caused by hard faults, silent errors add one degree of complexity, which is that failures manifest themselves only some time after the actual error occurred. So if the failure is detected, one might be tempted to roll back to the latest checkpoint, but one cannot be sure that the error occurred *before* the checkpoint was created, which means the checkpoint might be tainted. This means that checkpoint intervals would have to take the *mean time between (silent) errors* into account. Additionally, the user has to spend resources detecting the errors, and in many cases it might not be clear how to tell if certain data is corrupted. But for applications where this is possible, it can be useful to combine checkpointing with some verification mechanisms, as explained in [DHR15].

As with hard faults, replication has been used to detect and recover from silent errors, for example in [FME+12], where the authors make use of redundancy at the MPI level.

Application-specific detection and correction algorithms are also becoming increasingly popular for the same reasons as in the case of hard faults. Some recent examples where the numerical properties of the algorithms have been exploited to tolerate silent errors include:

- Using data analysis and time series to predict ranges of acceptable values in future time iterations [BBGD+15];
- Identifying invariants in linear solvers and using these to check for SDC (solvers include GMRES [EHM14a], CG, BiCG and Lanczos [Che13]);
- Using error bounds to detect errors in numerical integration solvers (for ordinary differential equations) [GZP+16];
- Computing a cheap, low order solution of a PDE with which to compare the high order (possibly corrupted) solution [BSS15].
- Using bounds of inner products appearing in optimization algorithms in computational chemistry (in particular, the Hartree-Fock method) [vDVDJ13].

It is however somewhat odd that a poorly understood problem is being studied in such detail. Often times, scientists make assumptions about the nature of silent errors that are not supported by data, such as assuming that silent errors will be predominantly caused by bit flips. In this thesis we want to study silent errors, so some words of caution are necessary.

2.3.2 A Methodology to Study Silent Errors

We follow the recommendations of Elliott et al. when thinking about how to study silent errors [EHM14b]. They identify some common mistakes made by authors in the field and propose some best practices.

Common Mistakes

There are now many papers out there where bit flips are injected into an algorithm to see how it reacts. As we already mentioned, the problem with this approach is that it is not clear that bit flips will be the main cause of the errors. Future hardware could behave in ways that give rise to other types of errors which we are currently unaware of.

Let us assume for a moment that bit flips *will* be the main cause of silent errors. What is the right way to simulate them? It is common practice in the research literature to inject bit flips randomly. Although this seems reasonable from a statistical point of view, injecting bit flips randomly only allows one to study the average behavior of an algorithm instead of the worst case scenarios, which algorithm designers should be most concerned with (by definition).

But even if we managed to understand *all* possible causes of silent errors (not only bit flips), we should not try to come up with different fault tolerance techniques for each type of fault. Not only would this be impractical, but it would be unnecessary. For a numerical algorithm it is not important to know what caused the error, but how the error is manifested. Since algorithms are designed in terms of error bounds, resilient algorithms should treat silent errors as numerical perturbations in the data, no matter the exact cause. As long as the error bounds are satisfied, the algorithm can be said to be resilient to silent errors.

One last thing to keep in mind is the effect of silent errors in the *metadata*. What if silent errors affect not only the floating-point data, but also pointers, counters or instructions? Well, corrupted metadata could manifest itself in three possible ways:

1. As an error in the floating-point data, for example, if a pointer points to a wrong address, retrieving a wrong value;

2. As a process fault, for example, if an index is out of bounds, causing a segmentation fault;

3. Keeping the process in operation but in an undefined state.

The first point takes us back to errors in the data. The second one would be equivalent to a hard fault, and we know how to deal with those. The third one seems to be extremely unlikely. This speaks in favor of focusing primarily on the floating-point data.

Best Practices

Classical ABFT approaches relying on checksums aim to guarantee the correctness of certain linear algebra computations. In light of our discussion above, it makes more sense to ensure that the numerical errors remain within the bounds admissible by the theory. In other words, one could admit corrupted data as long as the error is bounded. Elliott et al. call this *Skeptical Programming.*

Additionally, it is a good idea to identify which parts of the numerical algorithm *must* run reliably and in which parts can we allow errors to occur. The authors in [BFHH12] illustrate this concept (which they call *Selective Reliability*) with a fault tolerant GMRES solver, which performs an outer iteration of the Flexible GMRES algorithm coupled with an inner call to the classical GMRES. They then allow the inner GMRES solver to be unreliable, making sure that any errors are detected in the outer loop, which has to be reliable. This can save a lot of implementation effort, since not all parts of an algorithm have to be made fault tolerant.

Finally, in order to avoid missing the worst case scenarios by injecting bit flips randomly, one should focus on figuring out how the worst case scenarios might arise, which often translates into injecting silent faults of all possible orders of magnitude. Only then can one say that an algorithm is truly robust.

* * *

With these thoughts in mind, we are ready to take a look at the central algorithms for this thesis, namely the *sparse grid combination technique* and its variants, and see how the ideas discussed so far apply to them.

3 Sparse Grids

High-dimensional problems are commonplace in everyday life. Every shoe store owner knows the situation all too well: she would like to offer shoes for both men and women as well as different models. For each model she would like to have different colors and, for each color, different sizes. The warehouse quickly fills up as the number of possible pairs of shoes grows, draining the owner of all her initial capital.

This is an old problem in mathematics: adding a new dimension – model, color, size, etc. – to one's space of variables results in an exponential increase in the total number of degrees of freedom. This problem is commonly referred to as the *curse of dimensionality* and it is the subject of very active research.

High-dimensional problems appear in a wide range of fields. Shan and Wang [SW10] offer a good overview of the different strategies to tackle this class of problems and classify them into five major areas, summarized in Table 3.1.

In this thesis we will look at one of the most successful hierarchical decomposition methods, namely *sparse grids* [BG04]. Sparse grids aim to alleviate the curse of dimensionality by reducing the number of degrees of freedom hierarchically. The idea is to try to determine which degrees of freedom will a priori have

Strategy	**Examples**
Decomposition	Matrix formats, hierarchical and non-hierarchical schemes
Screening	Sensitivity analysis, ANOVA, PCA, optimization
Mapping	Artificial Neural Networks, fuzzy clustering, space-mapping
Space reduction	Adaptive response surface method, interval method, move-limit optimization, trust region algorithms
Visualization	Graph morphing, parallel coordinates, stacked displays

Table 3.1: A taxonomy of strategies for high-dimensional problems.

a small contribution to our solution space and eliminating them from our space of unknowns. This method (and its many variants) has been applied to a wide range of problems, from data mining [GGT01] to quantum mechanics [GH07]. Some initial concepts behind sparse grids were introduced in the sixties through the work of Smolyak [Smo63a], and later Zenger extended the ideas into what we currently understand as sparse grids [Zen91].

We now give a brief overview of sparse grids and one of its most useful variants: the sparse grid combination technique.

3.1 Basic Concepts

Let us first introduce the notation that we will use throughout this thesis. We first start by discretizing the unit interval $[0,1]$ using a one-dimensional grid, which we denote by Ω_l, where $l \in \mathbb{N}_+$. This grid has $2^l - 1$ inner points and may or may not have an additional grid point on each boundary. Since most of the examples we will encounter have boundary points, we will focus on this case. The grid Ω_l then has a total of $2^l + 1$ points and mesh size $h_l := 2^{-l}$. We denote each of the grid points in Ω_l as

$$x_{l,j} := j \cdot h_l, \quad 0 \leq j \leq 2^l.$$

In this notation we call the number l the *level* of the grid and j is simply an index that runs through the level.

When moving to d dimensions we use boldface letters to denote multi-indices, $\mathbf{l} =(l_1,\ldots,l_d) \in \mathbb{N}^d$. We discretize the d-unit cube using a d-dimensional full grid

$$\Omega_\mathbf{l} := \Omega_{l_1} \times \cdots \times \Omega_{l_d}.$$

The mesh sizes in each dimension can be written as

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

and the grid points in grid $\Omega_\mathbf{l}$ can be written as

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

Since we will be using multi-indices often, we should point out some common operations between them. Comparison operators are done component-wise. For example, two multi-indices $\mathbf{i}$ and $\mathbf{j}$ satisfy $\mathbf{i} \leq \mathbf{j}$ iff $i_k \leq j_k$ for all $k \in \{1, \ldots, d\}$. This entails that the strict inequality $\mathbf{i} < \mathbf{j}$ holds iff $\mathbf{i} \leq \mathbf{j}$ and $\mathbf{i} \neq \mathbf{j}$.

We will also apply discrete l_p-norms $|\cdot|_p$ to multi-indices, particularly

$$|\mathbf{l}|_1 := |l_1| + \cdots + |l_d|$$
$$|\mathbf{l}|_2 := \sqrt{l_1^2 + \cdots + l_d^2}$$
$$|\mathbf{l}|_\infty := \max_i |l_i|$$

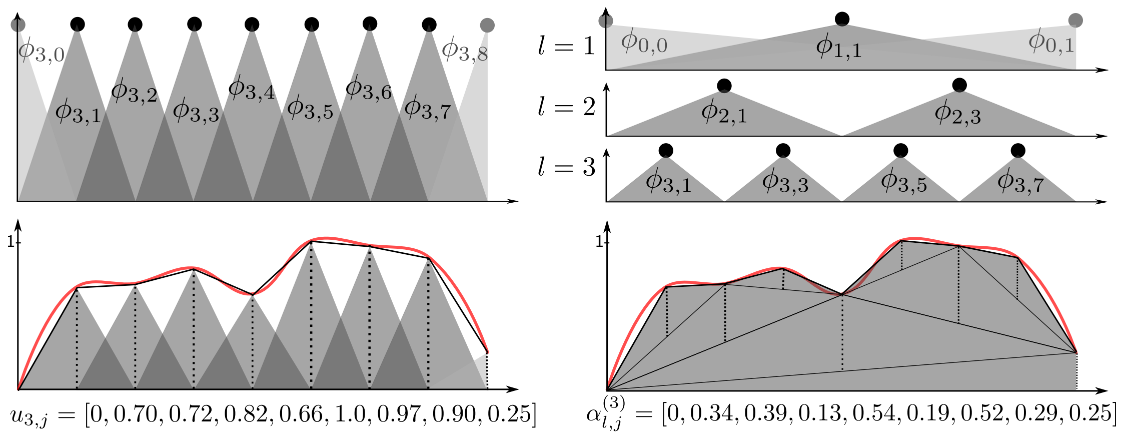


Figure 3.1: Nodal (left) and hierarchical (right) representations of a one-dimensional function of level $l = 3$. In the nodal basis we store the values of $u_{i,j}$, which correspond to the height of the nodal hat functions. In the hierarchical basis we store the surpluses $\alpha^{(i)}_{l,j}$, which represent the increments with respect to the previous level $l - 1$.

The wedge operator $\wedge$ will also be useful. $\mathbf{i} \wedge \mathbf{j}$ denotes the component-wise minimum of $\mathbf{i}$ and $\mathbf{j}$,

$$\mathbf{i} \wedge \mathbf{j} := (\min\{i_1, j_1\}, \dots, \min\{i_d, j_d\}).$$

We will also denote sets of multi-indices with capital calligraphic letters, for example $\mathcal{I}$, such that $\mathbf{l} \in \mathcal{I}$. A useful set of multi-indices is the *downset* $\mathcal{I}_\downarrow$ of a set $\mathcal{I}$, defined as

$$\mathcal{I}_\downarrow := \{\mathbf{l} \in \mathbb{N}^d : \exists \mathbf{k} \in \mathcal{I} \text{ s.t. } \mathbf{l} \leq \mathbf{k}\}.$$

In other words, the downset $\mathcal{I}_\downarrow$ contains all multi-indices smaller or equal to the multi-indices in $\mathcal{I}$.

Consider a function $u(\vec{x}) \in V \subset L^2([0,1]^d)$. We are interested in approximating this function in a discrete space, say $u_\mathbf{i}(\vec{x}) \in V_\mathbf{i} \subset V$, where $V_\mathbf{i} = \bigotimes_{k=1}^d V_{i_k}$ is the space of piece-wise d-linear functions defined on the grid $\Omega_\mathbf{i}$ [Gar13],

$$V_\mathbf{i} := \operatorname{span}\{\phi_{\mathbf{i},\mathbf{j}} \,:\, \mathbf{0} \leq \mathbf{j} \leq 2^\mathbf{i}\}. \tag{3.1}$$

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

$$\phi_{\mathbf{i},\mathbf{j}}(\vec{x}) := \prod_{k=1}^{d} \phi_{i_k,j_k}(x_k), \tag{3.2}$$

with

$$\phi_{i,j}(x) := \max(1 - |2^i x - j|, 0). \tag{3.3}$$

These hat functions allow us to interpolate $u_\mathbf{i}(\vec{x})$ on grid $\Omega_\mathbf{i}$ as follows:

$$u_\mathbf{i}(\vec{x}) = \sum_{\mathbf{0} \leq \mathbf{j} \leq 2^\mathbf{i}} u_{\mathbf{i},\mathbf{j}} \phi_{\mathbf{i},\mathbf{j}}(\vec{x}). \tag{3.4}$$

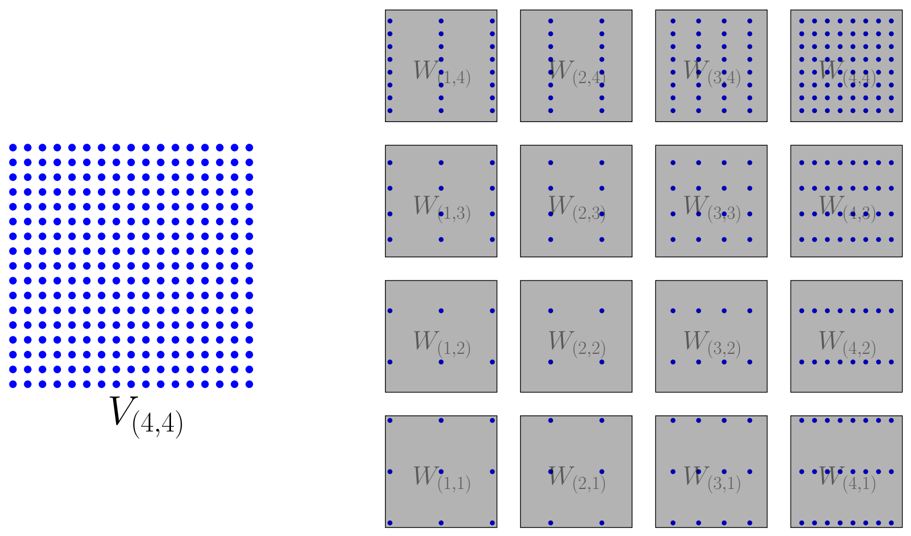


Figure 3.2: A full grid of level 4 in both dimensions (left) and the hierarchical subspaces that compose it (right).

We call Eq. (3.4) the *nodal representation* of $u_{\mathbf{i}}(\vec{x})$, and $u_{\mathbf{i},\mathbf{j}} \in \mathbb{R}$ are the *nodal coefficients*. Figure 3.1 (left) illustrates an interpolated function in the nodal representation, where the nodal coefficients correspond to the height of the hat functions $\phi_{\mathbf{i},\mathbf{j}}$. Note that we have added the basis functions $\phi_{0,0}$ and $\phi_{0,1}$ to account for the boundary points. A more detailed discussion of the different types of boundaries can be found in [Pfl10].

Now consider the *hierarchical spaces* $W_{\mathbf{l}}$ defined as

$$W_{\mathbf{l}} := \operatorname{span}\left\{\phi_{\mathbf{l},\mathbf{j}}(\vec{x}) : \mathbf{j} \in \mathcal{I}_{\mathbf{l}}\right\}, \tag{3.5}$$

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

$$\mathcal{I}_{\mathbf{l}} := \left\{\mathbf{j} : 1 \leq j_k \leq 2^{l_k} - 1, j_k \text{ odd}, 1 \leq k \leq d\right\}. \tag{3.6}$$

A hierarchical space $W_{\mathbf{l}}$ contains all functions $u_{\mathbf{i}} \in V_{\mathbf{i}}$ such that $u_{\mathbf{i}}$ vanishes on all grid points in the set $\bigcup_{\mathbf{l}<\mathbf{i}} \Omega_{\mathbf{l}}$ [H+15]. With the use of hierarchical spaces we can decompose a space $V_{\mathbf{i}}$ as follows:

$$V_{\mathbf{i}} = \bigoplus_{\mathbf{l}\leq\mathbf{i}} W_{\mathbf{l}}. \tag{3.7}$$

We illustrate this decomposition in two dimensions for $\mathbf{l} = (4,4)$ in Fig. 3.2.

Using the hierarchical spaces we can decompose a function $u_{\mathbf{i}} \in V_{\mathbf{i}}$ as follows:

$$u_{\mathbf{i}}(\vec{x}) = \sum_{\mathbf{l}\leq\mathbf{i}} h_{\mathbf{l}}(\vec{x}), \quad h_{\mathbf{l}}(\vec{x}) \in W_{\mathbf{l}} \tag{3.8}$$

$$= \sum_{\mathbf{l}\leq\mathbf{i}} \sum_{\mathbf{j}\in\mathcal{I}_{\mathbf{l}}} \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{i})} \phi_{\mathbf{l},\mathbf{j}}(\vec{x}). \tag{3.9}$$

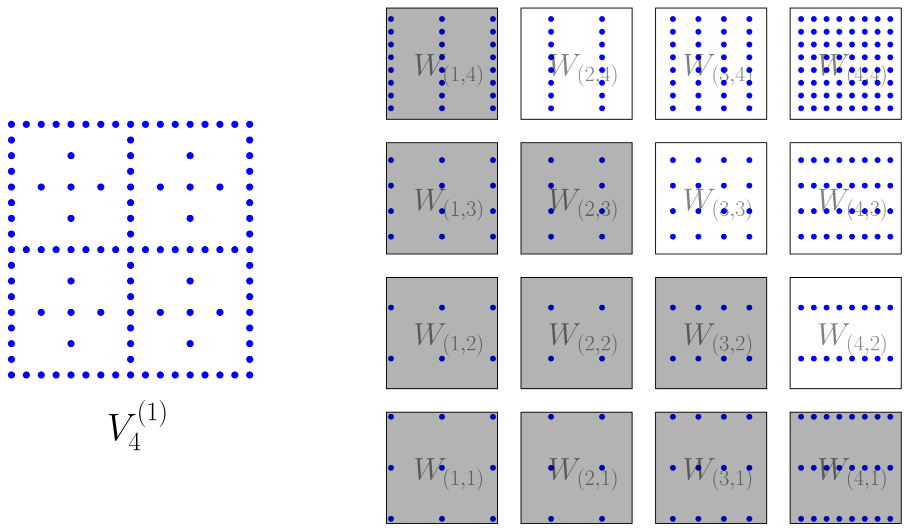


Figure 3.3: A classical sparse grid of level 4 (left) and the hierarchical subspaces that compose it (light gray, right).

We will call Eq. (3.9) the *hierarchical representation* of $u_{\mathbf{i}}(\vec{x})$. The coefficients $\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{i})} \in \mathbb{R}$ are called the *hierarchical coefficients* or *hierarchical surpluses*. If we have a list of function values in the nodal basis we can easily compute the hierarchical coefficients at the corresponding grid points. In one dimension they are obtained via the formula

$$\begin{aligned} \alpha_{l,j}^{(i)} &= u_i(x_{l,j}) - \frac{1}{2}(u_i(x_{l,j-1}) + u_i(x_{l,j+1})) \\ &= \left[-\tfrac{1}{2} \quad 1 \quad -\tfrac{1}{2}\right]_{l,j} u_i(x_{l,j}). \end{aligned} \tag{3.10}$$

In the second row of the equation we use the stencil notation for simplicity. The operation of going from the nodal to the hierarchical basis is called *hierarchization*. For a d-dimensional function we simply perform the hierarchization in each dimension:

$$\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{i})} = \left(\prod_{k=1}^{d} \left[-\tfrac{1}{2} \quad 1 \quad -\tfrac{1}{2}\right]_{l_k,j_k} \right) u_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}). \tag{3.11}$$

One can analogously compute the nodal coefficients if the hierarchical coefficients are known. This operation is commonly referred to as *dehierarchization*.

With these mathematical tools we can now address the curse of dimensionality. As we mentioned earlier, the problem is that a uniform d-dimensional grid $\Omega_{\mathbf{n}}$ has an exponentially large number of grid points ($|V_n| = h_n^{-d} = 2^{nd}$). The *classical sparse grid* space $V_n^{(1)} \subset V_n$ is defined as

$$V_n^{(1)} := \bigoplus_{|\mathbf{l}|_1 \le n+d-1} W_{\mathbf{l}}. \tag{3.12}$$

Compare the index boundary in the sum above with that of Eq. (3.7). This sum leaves out all spaces for which $n + d - 1 < |\mathbf{l}|_1 \le |\mathbf{n}|_1$. We have illustrated the resulting sparse grid space $V_n^{(1)}$ in Fig. 3.3 for $d = 2$ and $n = 4$. It ca nbe shown that this space has $|V_n^s| = \mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1})$ points [BG04], which is a dramatic reduction from the $|V_n| = \mathcal{O}(h_n^{-d})$ discretization points required by a full grid of the same level.

The rationale behind getting rid of all spaces $W_\mathbf{l}$ for which $n+d-1 < |\mathbf{l}|_1 \leq |\mathbf{n}|_1$ is as follows. Assume we want to interpolate a function u with bounded mixed second derivatives and homogeneous boundary conditions. Then one can show that the following estimates for the hierarchical components $h_\mathbf{l} \in W_\mathbf{l}$ hold:

$$\|h_\mathbf{l}\|_2 \leq 3^{-d} \cdot 2^{-2\cdot|\mathbf{l}|_1} \cdot |h|_{\mathbf{2},2}$$
$$\|h_\mathbf{l}\|_\infty \leq 2^{-d} \cdot 2^{-2\cdot|\mathbf{l}|_1} \cdot |h|_{\mathbf{2},\infty}$$

(And similarly for other norms. The details can be found in [BG04].) For our purposes it shall suffice to point out that the hierarchical components decay exponentially with increasing level $\mathbf{l}$. The question then becomes which subspaces one can get rid of without sacrificing much in accuracy, and an optimization analysis reveals that the splitting Eq. (3.12) is L_∞- and L_2-optimal (in some well-defined sense) [BG04]. Under these assumptions, the interpolation error on a sparse grid is [BG04]

$$\|u - u_n^{(1)}\|_2 = \mathcal{O}(h_n^2 \cdot (\log h_n^{-1})^{d-1}). \tag{3.13}$$

The error is thus only slightly larger than on a full grid, which is in $\mathcal{O}(h_n^2)$.

Some preliminary ideas behind sparse grids were first presented by Smolyak in the sixties [Smo63b] and then introduced formally by Zenger in the nineties in the context of partial differential equations [Zen91]. Since then, sparse grids have remained a topic of active research. We recommend the paper by Bungartz and Griebel as a general reference [BG04].

Despite the advantages that sparse grids offer in terms of computational requirements, it is not trivial to discretize and solve a PDE on a sparse grid. Furthermore, if one already has a robust solver that has been developed over many years and which is based on full grids, it might be infeasible to re-implement the solver to capture the sparse grid structure. In fact, we will see an example of such a code in Chapter 4. However, there is a way to profit from some of the properties of sparse grids without the need to discretize a domain directly on a sparse grid. This is the motivation behind the *combination technique.*

3.2 The Combination Technique

The combination technique was introduced by Griebel in the nineties to speed up the solution of PDEs on full grids [GSZ92, Gri92]. We illustrate the procedure with a simple time-dependent PDE, namely the two-dimensional linear advection equation:

$$\frac{\partial u}{\partial t} + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} = 0. \tag{3.14}$$

Suppose we are interested in solving this PDE 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

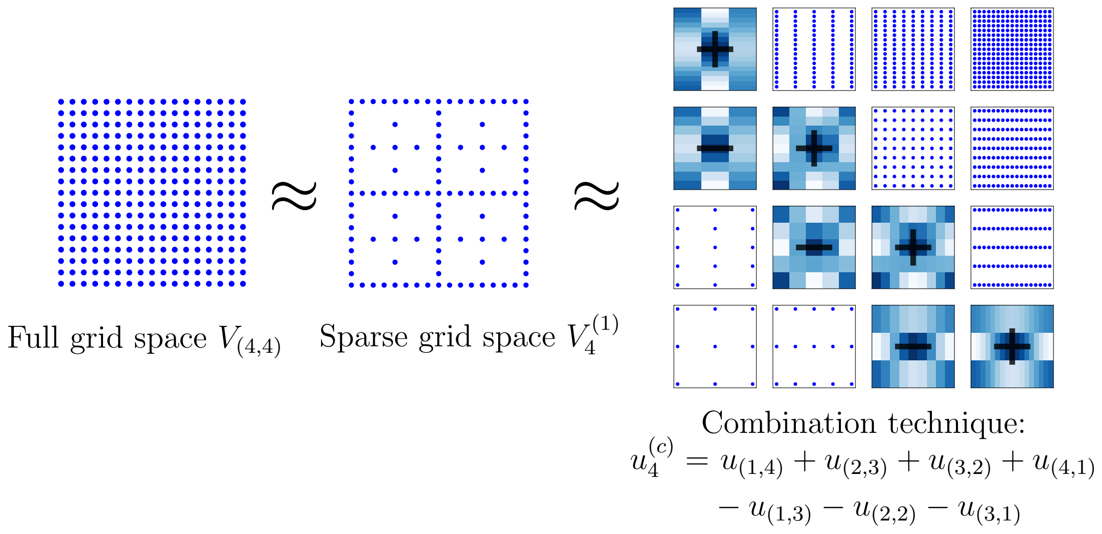

Full grid space $V_{(4,4)}$
$\approx$
Sparse grid space $V_4^{(1)}$
$\approx$
Combination technique:
$u_4^{(c)} = u_{(1,4)} + u_{(2,3)} + u_{(3,2)} + u_{(4,1)} - u_{(1,3)} - u_{(2,2)} - u_{(3,1)}$

Figure 3.4: Illustration of the classical combination technique (Eq. (3.16)) for the linear advection equation in 2D with $n = 4$.

conditions. Here, c_x and c_y are real positive constants defining the advection velocity. The solution of Eq. (3.14) is simply the initial condition translated by the advection velocity,

$$u(x, y, t) = \sin(2\pi(x - c_x t)) \sin(2\pi(y - c_y t)).$$

Let us now suppose we are given a black box solver for Eq. (3.14) which receives as input at least the number of discretization points in the x and y directions. In our case we have an implementation of a Lax-Wendroff scheme [Win11]. In Fig. 3.4 (left) we have plotted the solution of Eq. (3.14) at time $t = 0.5$ with velocities $c_x = c_y = 0.5$ on a full grid $\Omega_{\mathbf{n}}$ of level $\mathbf{n} = (4, 4)$ (meaning it has $(2^4+1)\times(2^4+1)$ grid points). Now imagine that this level of resolution is too expensive and cannot be afforded. Re-discretizing the domain on a sparse grid is not an option in this case since we assume that the black box solver cannot be altered or that it would take too much effort. What one can do instead is to try to approximate the full grid solution by solving the PDE on several coarser, anisotropic grids instead, and then *combining* the results together. This has been illustrated in Fig. 3.4 (right). Here, instead of solving the advection equation on grid $\Omega_{(4,4)}$ we solve it on seven different grids, namely, $\Omega_{(1,4)}$, $\Omega_{(1,3)}$, $\Omega_{(2,3)}$, $\Omega_{(2,2)}$, $\Omega_{(3,2)}$, $\Omega_{(3,1)}$ and $\Omega_{(4,1)}$. Notice that these grids have considerably fewer points than $\Omega_{(4,4)}$ (either one eighth of one sixteenth of the points). The idea of the combination technique is to combine these solutions with the weights as indicated in the figure in the hope that the result is close to that of the full grid.

The connection to sparse grids can be seen by noticing that the combined solution lives on the the sparse grid space $V_4^{(1)}$. The choice of the combination coefficients (in this case $+1$ and -1) is what ensures that the combination approximates the solution in the sparse grid space. One can incrementally approximate the solution in the space $V_4^{(1)}$ by starting with, say, $u_{(1,4)} \in V_{(1,4)} = \bigoplus_{\mathbf{l}\le(1,4)} W_{\mathbf{l}}$.

This approximation can be then improved by adding the function $u_{(2,3)}$, which adds the contributions from all spaces $W_{\mathbf{l}}$ with $\mathbf{l} \leq (2,3)$. But the result of $u_{(1,4)} + u_{(2,3)}$ has *twice* the contributions from subspaces $W_{\mathbf{l}}$ with $\mathbf{l} \leq (1,4) \wedge (2,3) = (1,3)$. The solution is then to subtract the function that contains exactly those spaces, namely $u_{(1,3)}$. This can be thought of as applying an inclusion-exclusion principle. The combination technique simply continues this procedure until only the hierarchical spaces corresponding to the sparse grid are left.

Let us now write down the combination technique in its most general form. We approximate the sparse grid solution $u_n^{(1)}$ (and thus the full grid solution u_n) full grid solution by a *combination solution* $u_n^{(c)}$ given by

$$u_{\mathbf{n}} \approx u_n^{(1)} \approx u_n^{(c)} = \sum_{\mathbf{i} \in \mathcal{I}} c_{\mathbf{i}} u_{\mathbf{i}}. \tag{3.15}$$

The weights $c_{\mathbf{i}} \in \mathbb{R}$ are called *combination coefficients*. The solutions $u_{\mathbf{i}}$ are typically called *component solutions* and the grids $\Omega_{\mathbf{i}}$ *component grids*. We will often refer to the sparse grid that results from the combination technique as the *combination grid* $\Omega_n^{(c)}$. $\mathcal{I}$ is a set of multi-indices.

How well the combination technique approximates the full grid solution depends on how the coefficients $c_{\mathbf{i}}$ and the set $\mathcal{I}$ are chosen, since not all choices will give reasonable results. In our example we introduced the *classical combination technique* which is given by

$$u_n^{(c)} = \underbrace{\sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q}}_{=c_{\mathbf{i}}} \sum_{\mathbf{i} \in \mathcal{I}_q^{d,n}} u_{\mathbf{i}}. \tag{3.16}$$

The index set is defined as

$$\mathcal{I}_q^{d,n} = \{\mathbf{i} : |\mathbf{i}|_1 = n + (d-1) - q\}. \tag{3.17}$$

The combination coefficients $c_{\mathbf{i}}$ are simply the binomial coefficients with alternating sign, which ensures that the combination satisfies the inclusion-exclusion principle.

Our example was generated with the choice $d = 2$ and $n = 4$, yielding the sets

$$\mathcal{I}_0^{2,4} = \{(1,4),(2,3),(3,2),(4,1)\},$$
$$\mathcal{I}_1^{2,4} = \{(1,3),(2,2),(3,1)\}.$$

For a general combination of the form (3.15), the combination coefficients can be calculated as

$$c_{\mathbf{i}} = \sum_{\mathbf{i} \leq \mathbf{j} \leq \mathbf{i}+\mathbf{1}} (-1)^{|\mathbf{j}-\mathbf{i}|} \chi_{\mathcal{I}}(\mathbf{j}), \tag{3.18}$$

where $\chi_{\mathcal{I}}$ is the indicator function of set $\mathcal{I}$ [Har15].

The component solutions are combined either by interpolating them to the full grid space or by using hierarchization. When switching to the hierarchical basis one can directly add the corresponding surpluses $\alpha^{(\mathbf{i})}_{\mathbf{l},\mathbf{j}}$ in the sparse grid space [HP16a, Hup13]. We will say a few more words about the combination step in Chapter 4, but before going any further, we discuss the convergence properties of the combination techniques and the costs involved.

3.2.1 Convergence and Costs of the Combination Technique

The theory of the classical combination technique as presented by Griebel establishes that the combination technique converges if each of the component solutions $u_{\mathbf{i}}$ satisfies the *error splitting assumption* (ESA) [GSZ92]. In arbitrary dimensions the ESA is given by [Har15]

$$u - u_{\mathbf{i}} = \sum_{k=1}^{d} \sum_{\substack{\{e_1,\dots,e_k\} \\ \subset\{1,\dots,d\}}} C_{e_1,\dots,e_k}(\vec{x}, h_{i_{e_1}}, \dots, h_{i_{e_k}}) h^p_{i_{e_1}} \cdots h^p_{i_{e_k}}, \tag{3.19}$$

where $p \in \mathbb{N}$ is the order of the discretization scheme; the $C_{e_1,\dots,e_k}(\vec{x}, h_{i_{e_1}}, \dots, h_{i_{e_k}})$ are functions depending on the spatial coordinates $\vec{x}$ and on the different mesh sizes $h_{\mathbf{i}}$, and they should bounded:

$$|C_{e_1,\dots,e_k}(\vec{x}, h_{i_{e_1}}, \dots, h_{i_{e_k}})| \leq \kappa_{e_1,\dots,e_k}(\vec{x}), \quad \forall \{e_1,\dots,e_k\} \subset \{1,\dots,d\}.$$

At the same time, the functions $\kappa_{e_1,\dots,e_k}$ should also be bounded by

$$\kappa_{e_1,\dots,e_k}(\vec{x}) \leq \kappa(\vec{x}).$$

It is important to note that Eq. (3.19) holds *point-wise*, which means that it must hold for all points $\vec{x}$ independently. This can be seen by the explicit dependence of each function $C_{e_1,\dots,e_k}$ on $\vec{x}$.

In one dimension the ESA reduces to

$$u - u_i = C_1(x_1, h_i) h^p_i, \quad |C_1(x_1, h_i)| \leq \kappa_1(x_1). \tag{3.20}$$

In two dimensions one has

$$u - u_{\mathbf{i}} = C_1(x_1, x_2, h_{i_1}) h^p_{i_1} + C_2(x_1, x_2, h_{i_2}) h^p_{i_2} + C_{1,2}(x_1, x_2, h_{i_1}, h_{i_2}) h^p_{i_1} h^p_{i_2}. \tag{3.21}$$

In other words, the error expansion is the sum of the univariate contributions corresponding to the mesh sizes h_i and a multivariate cross term. Griebel showed for the two- and three-dimensional cases that if such a point-wise expansion holds, then the error of the combination technique is asymptotically equal to that of the corresponding sparse grid solution,

$$\|u - u^{(c)}_n\|_2 = \mathcal{O}(h^2_n \cdot (\log h^{-1}_n)^{d-1}). \tag{3.22}$$

This holds for arbitrary dimensions, as was shown by Reisinger [Rei12].

The price to pay for having the accuracy of a sparse grid and the convenience of dealing only with full anisotropic grids is that we have many such component grids. The number of component grids in the classical combination technique is [Pfl10]

$$\left|\bigcup_{q=0}^{d-1} \mathcal{I}_q^{d,n}\right| = \mathcal{O}(d(\log h_n^{-1})^{d-1}). \tag{3.23}$$

Each of the component grids has in turn

$$|\Omega_{\mathbf{i}}| = \mathcal{O}(h_n^{-1})$$

grid points, bringing the total cost of the combination technique to

$$\mathcal{O}(d(\log h_n^{-1})^{d-1} h_n^{-1}).$$

But this is where we can make use of parallel systems to our great advantage, turning the extra effort of the combination technique into an opportunity for parallelization beyond domain decomposition. In many cases, the PDE can be solved on each of the component grids independently of each other, and then combining the results by communicating among the processing elements. We will go into further detail about the parallelization in Chapter 4.

We now introduce a useful variant of the classical combination technique.

3.2.2 The Truncated Combination Technique

Often the error of the combination technique is dominated by the most anisotropic component grids. For this reason it helps to introduce a truncation parameter to exclude these highly anisotropic grids. The *truncated combination technique* is given by [Har16b]

$$u_{n,\boldsymbol{\tau}}^{(c)} = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\mathbf{i} \in \mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}} u_{\mathbf{i}}, \tag{3.24}$$

with the index set

$$\mathcal{I}_{q,\boldsymbol{\tau}}^{d,n} = \{\mathbf{i} : |\mathbf{i}|_1 = n + (d-1) + |\boldsymbol{\tau}|_1 - q, \quad \mathbf{i} > \boldsymbol{\tau}\}. \tag{3.25}$$

The truncation parameter $\boldsymbol{\tau}$ allows us to define a shift in the set of indices. If $\boldsymbol{\tau} = \mathbf{0}$ we have the classical combination technique, but for any $\boldsymbol{\tau} \geq \mathbf{0}$ we force a minimum level of resolution in every dimension. For example, setting $\boldsymbol{\tau} = (1,1)$ in our example we obtain the following index sets:

$$\mathcal{I}_{0,\mathbf{1}}^{2,4} = \{(2,5),(3,4),(4,3),(5,2)\},$$

$$\mathcal{I}_{1,\mathbf{1}}^{2,4} = \{(2,4),(3,3),(4,2)\}.$$

Notice that we now approximate the full grid solution $u_{\mathbf{n}'}$ with $\mathbf{n}' = \mathbf{n} + \boldsymbol{\tau}$. To approximate the original solution $u_{(4,4)}$ using truncation (say, $\boldsymbol{\tau} = (1,1)$) one

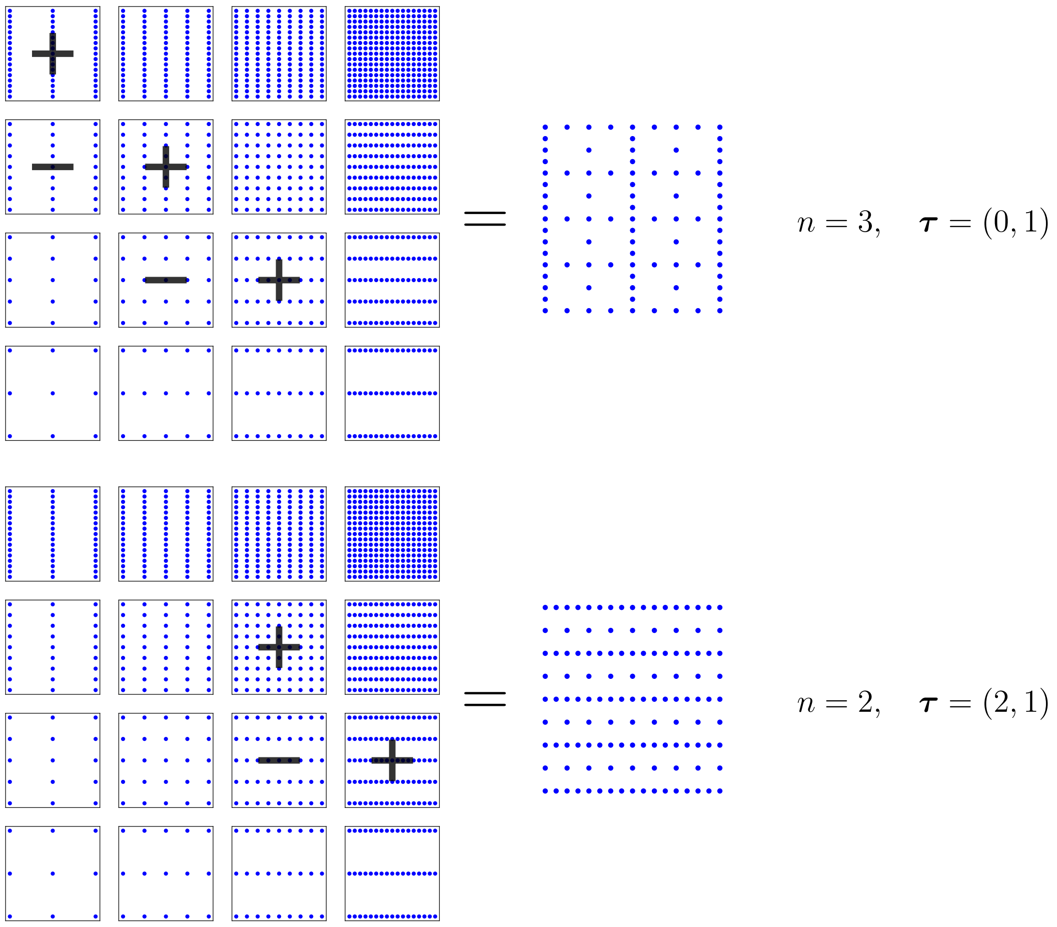

$n = 3, \quad \boldsymbol{\tau} = (0,1)$
$n = 2, \quad \boldsymbol{\tau} = (2,1)$

Figure 3.5: Two examples of the truncated combination technique with different resolutions in each dimension, Eq. (3.24).

should set $n = 3$ (which, shifted by $\boldsymbol{\tau}$ gives the desired full grid level $(4,4)$). This would result in the following index sets:

$$\mathcal{I}^{2,3}_{0,\mathbf{1}} = \{(2,4),(3,3),(4,2)\},$$

$$\mathcal{I}^{2,3}_{1,\mathbf{1}} = \{(2,3),(3,2)\}.$$

In this case we have fewer component grids but each of them is twice as expensive as the original grids.

For convenience we will denote as $\mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ the index set with all combination indices,

$$\mathcal{I}^{d,n}_{\boldsymbol{\tau}} := \bigcup_{q=0}^{d-1} \mathcal{I}^{d,n}_{q,\boldsymbol{\tau}} \tag{3.26}$$

This will be the most common formulation of the combination technique throughout this thesis, so to give a better idea of how it looks we illustrate two examples in Fig. 3.5. Below, in Table 3.2, we see how the error behaves depending on the choice of the truncation parameter $\boldsymbol{\tau}$. The higher the truncation for a constant

$\boldsymbol{\tau}$	n	# grids	$e = \frac{\lVert u_{n,\boldsymbol{\tau}}^{(c)} - u_{\text{exact}} \rVert_2}{\lVert u_{\text{exact}} \rVert_2}$
(0,0)	5	9	2.07×10^{-2}
(1,1)	4	7	1.19×10^{-2}
(2,2)	3	5	5.24×10^{-3}
(3,3)	2	3	3.12×10^{-3}
Full grid, $\mathbf{n} = (5,5)$			3.24×10^{-3}

Table 3.2: Error of different truncated combination techniques for the linear advection equation.

full grid level $\mathbf{n}'$, the better the approximation (but each solution in the resulting combination is more expensive).

Algorithm 1 describes the basic steps to solve a time-dependent PDE using the truncated combination technique. One first sets the initial conditions (line 3), after which the PDE solver is called for each index (line 6). The solver then evolves the initial condition for a certain number of time steps N_t, and once this is done, the component solution is transformed to the hierarchical basis (line 7). This is necessary for the combination step (line 8). The combined solution can then be transformed back to the nodal basis (line 9) so that each component solution can be updated to the value of the combined solution. This process can be repeated until a certain convergence criterion is reached or after a certain number of total time steps have been performed.

The combination technique has been used to address a wide variety of problems, ranging from option pricing [RW07] and machine learning [Gar07], all the way to plasma physics [KH13] and quantum mechanics [GG00].Additionally, other variants of the combination technique have been developed to solve a wider range of problems. They differ in how the combination coefficients $c_{\mathbf{i}}$ or the index set $\mathcal{I}$ are chosen, and the resulting combinations vary in approximation quality. Three notable examples are dimension-adaptive sparse grids [Heg03], combinations based on multivariate extrapolation [Har15] and the optimized combination technique [HGC07a], which we briefly describe now, since it has also been used in the context of fault tolerance.

3.3 The Fault Tolerant Combination Technique

In Chapter 2 we looked at the different types of faults that can affect an HPC system, and in this chapter we introduced a numerical method to solve high-dimensional PDEs and argued that this algorithm can be run in parallel. At this point it is possible to introduce a variant of the combination technique that can be used in case both hard and soft faults occur without saying anything about the parallelization strategy. We need only assume that subsets of component solutions

Algorithm 1: Truncated combination technique to solve time-dependent PDEs

```
input  : A function solver; sparse grid resolution $n$; parameter $\boldsymbol{\tau}$; time
         steps per combination $N_t$
output: Combined solution $u^{(c)}_{n,\boldsymbol{\tau}}$
1  Generate index set $\mathcal{I}^{d,n}_{\boldsymbol{\tau}} = \bigcup_{q=0}^{d-1} \mathcal{I}^{d,n}_{q,\boldsymbol{\tau}}$ and compute coefficients $c_{\mathbf{i}}$;
2  for $\mathbf{i} \in \mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ do
3      $u_{\mathbf{i}} \leftarrow u(\vec{x}, t = 0)$ ;                          // Set initial conditions
4  while not converged do
5      for $\mathbf{i} \in \mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ do
6          $u_{\mathbf{i}} \leftarrow$ solver($u_{\mathbf{i}}$,$N_t$) ;       // Solve the PDE on grid $\Omega_{\mathbf{i}}$ ($N_t$ time steps)
7          $u_{\mathbf{i}} \leftarrow$ hierarchize($u_{\mathbf{i}}$) ;          // Transform to hier. basis, Eq. (3.9)
8      $u^{(c)}_{n,\boldsymbol{\tau}} \leftarrow$ combine($c_{\mathbf{i}}u_{\mathbf{i}}$) ;         // Combined solution (in the hier. basis)
9      $u^{(c)}_{n,\boldsymbol{\tau}} \leftarrow$ dehierarchize($u^{(c)}_{n,\boldsymbol{\tau}}$) ;          // Transform back to nodal basis
10     for $\mathbf{i} \in \mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ do
11         $u_{\mathbf{i}} \leftarrow$ sample($u^{(c)}_{n,\boldsymbol{\tau}}$) ;             // Sample each $u_{\mathbf{i}}$ from new $u^{(c)}_{n,\boldsymbol{\tau}}$
```

are computed independently of each other, so that a system failure will affect only a subset of all solutions.

Consider the example shown in Fig. 3.6 where we try to solve once again the advection equation on seven different component grids. As we will see in Chapter 5, the vast majority of the computation time in a parallel implementation of the combination technique is spent solving the PDE on the component grids. This is true even when one chooses to combine the results not only once at the end of the computation but maybe every certain number of time steps (or even after every time step). This means that if the parallel system is affected by faults, they will most likely occur during the computation phase. Figure 3.6 (left) shows how a system fault could affect a subset of the component solutions, in this case $u_{(2,2)}$ and $u_{(2,3)}$. We will denote by $\mathcal{J}$ the set of the multi-indices corresponding to the affected solutions.

As we discussed in Chapter 2, one strategy to deal with faults is to perform checkpoints and restart the solution from the last checkpoint. One possible way to do this with the combination technique would be to checkpoint the combined (sparse grid) solution and restart the failed component solutions from the last checkpoint. But this approach is hardly feasible in a parallel environment, as argued by Harding, who gives the following example. If the system has 100 processing elements (say, nodes) available and one of them fails and tries to recompute a component solution from the last checkpoint, then the other 99 nodes will be idle during this time, bringing the parallel efficiency to at most 1% during that time [Har16b]. Of course, one could try to overlap the recomputation time with the actual computation time to bring down the overhead, but the worst case scenario just presented could still happen in the general case. Additionally, as fault rates increase in larger systems, it becomes unclear whether such an approach can scale.

To overcome this difficulty, Harding et al. developed the *fault tolerant combination technique* [H+15]. The basic idea is to find an alternative combination of component solutions such that one does not use the data that was lost due to faults. This idea is illustrated once again with a simple advection equation in Fig. 3.6. On the left we see a classical combination technique with seven component solutions. The two solutions marked with red crosses are assumed to have failed either partly or entirely at some point during the computation, so the corresponding data goes lost. On the right we show an alternative combination of partial solutions, this time with only five component grids. Notice that this combination excludes the two failed component solutions $u_{2,3}$ and $u_{2,2}$ but adds a new component solution that was not originally in the combination, namely $u_{1,2}$. We will later talk about this in more detail.

We now try to address some of the most immediate questions that arise at this point, namely,

- How do we choose a new combination of solutions that guarantees that failed solutions will be excluded?
- How much effort does it take to find such an alternative combination?
- What are the costs and approximation errors involved?

To answer these questions we follow the ideas from [H+15].

We are looking for general combinations of the form (3.15) whose coefficients $c_{\mathbf{l}}$

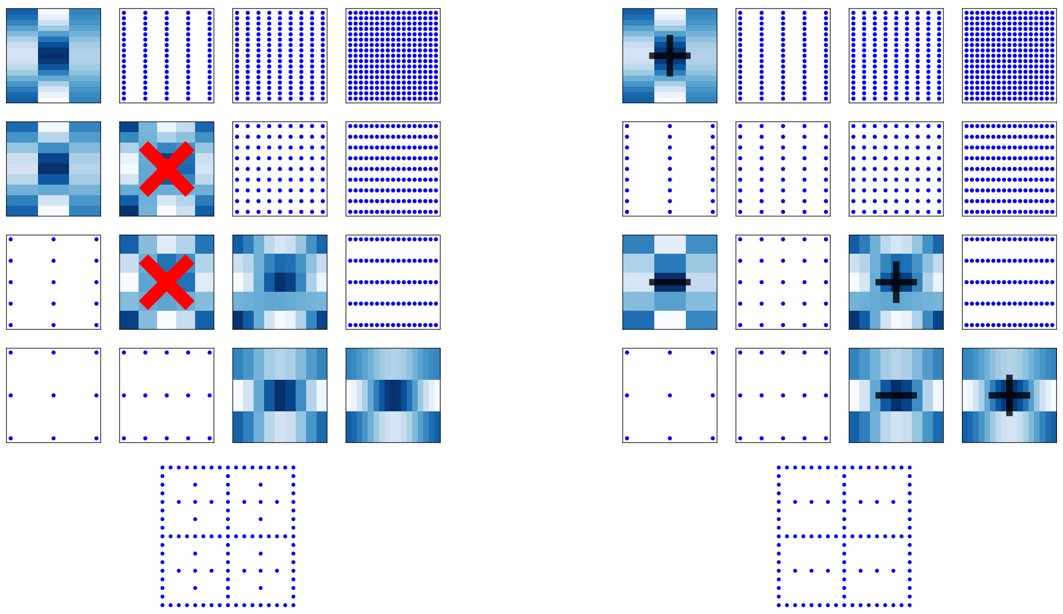
1. satisfy the inclusion-exclusion principle, which is necessary to have consistent combinations, and
2. minimize a given sparse grid interpolation error.

The inclusion-exclusion principle can be fulfilled if, for each coefficient in the combination, we have

$$\sum_{\mathbf{k}\in\mathcal{I},\mathbf{k}\geq\mathbf{i}} c_{\mathbf{k}} \in \{0,1\} \quad \text{for } \{c_{\mathbf{i}}\}_{\mathbf{i}\in\mathcal{I}}. \tag{3.27}$$

For the second point we need to assume that the real solution u lives in the Sobolev space H^2_{mix} equipped with the semi-norm $\|u\|_{H^2_{\text{mix}}} := \left\|\frac{\partial^{2d}}{\partial x_1^2\cdots\partial x_d^2}u\right\|_2^2$, which means it has bounded mixed derivatives, i.e., u has bounded mixed derivatives up to second order. If we examine the interpolation error of the combination technique solution $u_n^{(c)}$ with respect to u, we see that the error is bounded by

$$\|u - u_n^{(c)}\|_2 \;\leq\; 3^{-d}\|u\|_{H^2_{\text{mix}}} \sum_{\mathbf{i}\in\mathbb{N}^d} \left(4^{-\|\mathbf{i}\|_1} \left| 1 - \sum_{\mathbf{k}\in\mathcal{I},\mathbf{k}\geq\mathbf{i}} c_{\mathbf{k}} \right| \right), \tag{3.28}$$



$$\begin{aligned} u_n^{(c)} &= u_{(1,4)} + \cancel{u_{(2,3)}} + u_{(3,2)} + u_{(4,1)} \\ &\quad - u_{(1,3)} - \cancel{u_{(2,2)}} - u_{(3,1)} \end{aligned}$$

$$\begin{aligned} u_n^{(c)} &= u_{(1,4)} + u_{(3,2)} + u_{(4,1)} \\ &\quad - u_{(1,2)} - u_{(3,1)} \end{aligned}$$

Figure 3.6: Left: System fault affects two component solutions either entirely or partially. Right: New combination coefficients that exclude failed solution. The resulting sparse grid has fewer points than the original, which means that the approximation quality of the combination is slightly worse.

We can then define the quantity

$$Q(\{c_{\mathbf{i}}\}_{\mathbf{i}\in\mathcal{I}}) := \sum_{\mathbf{i}\in\mathcal{I}_\downarrow} 4^{-\|\mathbf{i}\|_1} \sum_{\mathbf{k}\in\mathcal{I},\mathbf{k}\geq\mathbf{i}} c_{\mathbf{k}}, \tag{3.29}$$

and observe that Eq. (3.28) is minimized when Q is maximized with constraints $\sum_{\mathbf{k}\in\mathcal{I},\mathbf{k}\geq\mathbf{i}} c_{\mathbf{k}} = \{0,1\}$. At this point it is useful to define the *hierarchical coefficient* as the quantity

$$w_{\mathbf{i}} := \sum_{\mathbf{k}\in\mathcal{I},\mathbf{k}\geq\mathbf{i}} c_{\mathbf{k}} \in \{0,1\}, \quad \mathbf{i}\in\mathcal{I}_\downarrow. \tag{3.30}$$

The equation above gives rise to a linear system of equations which can be compactly written as

$$\vec{w} = M\vec{c}. \tag{3.31}$$

The vectors $\vec{w}$ and $\vec{c}$ contain all coefficients $w_{\mathbf{i}}$ and $c_{\mathbf{i}}$ (in any given ordering), and M is the system matrix of size $|\mathcal{I}_\downarrow| \times |\mathcal{I}_\downarrow|$. This notation allows us to write the maximization problem as

$$\begin{aligned} \max_{\mathbf{w}} Q'(\mathbf{w}), \quad \text{s.t.}\ & w_{\mathbf{i}} \in \{0,1\} \quad \forall \mathbf{i}\in\mathcal{I}_\downarrow, \\ & c_{\mathbf{i}} = (M^{-1}\vec{w})_{\mathbf{i}} = 0 \quad \forall \mathbf{i}\in\mathcal{I}_\downarrow\backslash\mathcal{I}, \end{aligned} \tag{3.32}$$

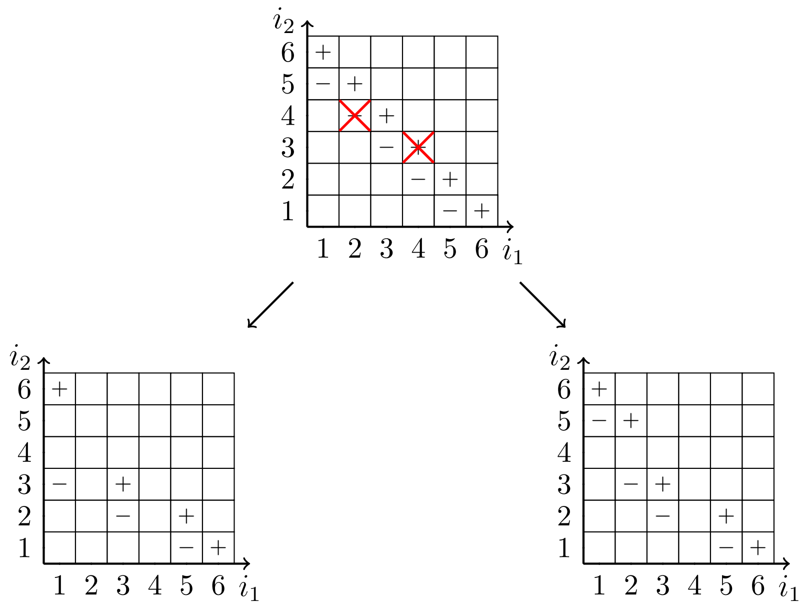


Figure 3.7: Depending on which component solutions fail, there can be different possible combinations that exclude them. Here we show two combinations that satisfy both constraints in Eq. (3.32), but the bottom right combination maximizes Q'.

where

$$Q'(\vec{w}) := \sum_{\mathbf{i}\in\mathcal{I}_\downarrow} 4^{-\|\mathbf{i}\|_1} w_{\mathbf{i}}. \tag{3.33}$$

This maximization problem is called the *generalized coefficient problem* (GCP). Notice that the difference between the quantities Q and Q' is simply that for Q' we consider all coefficients $c_{\mathbf{l}}$ in the downset $\mathcal{I}_\downarrow$ instead of only in $\mathcal{I}$, which is why the second condition $c_{\mathbf{l}} = 0$ has to be added for the coefficients that do not belong to $\mathcal{I}$.

The steps to find new combination coefficients when faults occur would look as follows:

1. Identify indices of failed component solutions $u_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{J}$ and remove them from the index set $\mathcal{I}$.
2. Generate list of $\vec{w}$ vectors for which the constraint $(M^{-1}\vec{w})_{\mathbf{i}} = 0$ holds true.
3. For each vector $\vec{w}$, calculate $Q'(\vec{w})$ and choose the one that maximizes Q'.
4. Calculate new combination coefficients $\vec{c} = M^{-1}\vec{w}$.

Figure 3.7 illustrates one more combination scheme with two faults $\{(2,4),(4,3)\}$ and two possible alternative combinations for which the conditions $c_{(2,4)} = c_{(4,3)} = 0$ are satisfied. The lower right combination, however, results in a larger value of Q' and should therefore be preferred to the one on lower left one. Something else worth noticing is that in both cases we need new component solutions that were not part of the original combination technique. In the left case these would be $u_{1,3}$ and $u_{3,2}$. These should be available during the combination step, which means that they should be computed in advance along with the rest of the component solutions. For example, in a two-dimensional combination technique we would need *two* additional index sets $\mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$ with $q = d$ and $q = d + 1$, since in the worst case scenario, all combination solutions would fail, which would result in an alternative combination technique two levels smaller.

As we go to higher dimensions things become more complicated, since the steps we described to compute the alternative combination coefficients cannot be carried out straightforwardly. This is because the maximization problem (3.32) is NP-hard, which in practical terms means that for an arbitrary set of faults in any dimension, there can be exponentially many possible solutions to the GCP and they cannot be all be investigated. But an important observation is that the time required to solve the GCP strongly depends on which component solutions are affected. More specifically, if the failed component solutions belong to a set $\mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$ with a higher value of q, the GCP takes longer to solve. Equivalently, if the failed component solutions belong to a set with $q = 0$ or $q = 1$ the GCP can be solved quickly.

Since we cannot afford to spend too much computing time finding new combination coefficients, Harding et al. propose the following compromise:

- Independently of the dimension, always add two sets of component solutions corresponding to the index sets $\mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$ with $q = d$ and $q = d + 1$.
- If faults affect component solutions for which $q = 0$ or $q = 1$, solve the GCP to find a new combination.
- All other failed component solutions ($q = 2, \ldots, d-1$) *should be recomputed.*

This approach strikes a balance between the complexity of finding new combination coefficients and the cost of recomputing some component solutions. Since the component solutions with $q = 2, \ldots, d-1$ are considerably less expensive than those with $q = 0, 1$ we can afford to recompute them if they fail. The authors in [H$^+$15] have shown (using results from [HH13]) that the overhead of the FTCT, as measured by the additional grid points needed for the extra index sets is given by the following

Proposition: 3.3.1 *Let $|\Omega_{\boldsymbol{i}}|$ denote the number of grid points in $\Omega_{\boldsymbol{i}}$, and let N_n and L_n be the total number of grid points in the collection of grids with indices in $\mathcal{L}^{d,n} = \bigcup_{q=0}^{d-1} \mathcal{I}_{q,\mathbf{0}}^{d,n}$ and $\mathcal{N}^{d,n} = \bigcup_{q=0}^{d+1} \mathcal{I}_{q,\mathbf{0}}^{d,n}$ respectively, that is,*

$$L_n := \sum_{\mathbf{i} \in \mathcal{L}^{d,n}} |\Omega_{\boldsymbol{i}}|\,,$$

$$N_n := \sum_{\mathbf{i} \in \mathcal{N}^{d,n}} |\Omega_{\boldsymbol{i}}|\,.$$

Then the fraction χ_n of additional grid points required by the two extra index sets of the FTCT is given by

$$\chi_n := \lim_{n\to\infty} \frac{N_n - L_n}{L_n} = \frac{3}{4(2^d-1)}.$$

We want to emphasize that the FTCT yields good results for the same reason sparse grids can cope well with the curse of dimensionality in a large spectrum of problems: once we identify different levels of hierarchy, we only sacrifice the information on the highest hierarchical level, whose contribution is a priori the smallest. Classical sparse grids are optimal in the sense that they get rid of just enough hierarchical subspaces as to retain an acceptable quality. Removing further subspaces would result in a loss of quality that is larger than the benefit of having fewer subspaces. The FTCT incurs this suboptimal trade-off, but we agree to pay this price in order to be fault tolerant.

One final optimization hint. One can speed up the computation of the new combination coefficients if the set of fault indices $\mathcal{J}$ can be divided into smaller disjoint subsets, i.e., if faults occur "far away" from each other in the 1-norm. That is, if two fault indices $\mathbf{i}, \mathbf{j} \in \mathcal{J}$ are sufficiently far from each other, one can solve two independent GCPs, one for $\mathbf{i}$ only and one for $\mathbf{j}$ only, both of which are easier to solve than a single GCP with both faults. The criterion to decide if two faults are sufficiently far away from each other is if the set of all indices in $\mathcal{I}$ larger than $\mathbf{i}$ does not overlap with the set of all indices in $\mathcal{I}$ larger than $\mathbf{j}$, i.e., if

$$\mathbf{k} \notin \mathcal{I}, \quad k_l = \max_l\{i_l, j_l\}, \quad l = 1, \dots, d.$$

In Chapter 5 we will show that experiments confirm this tradeoff when we put the fault tolerant combination technique to test with five-dimensional plasma simulations.

Fault Tolerance with the Optimized Combination Technique (OptiCom)

Before moving on, we want to briefly describe an alternative way to use the combination technique to be tolerant to faults. It is based on the so-called optimized combination technique, or OptiCom [HGC07b], which attempts to find combination coefficients $\mathbf{c}$ a posteriori – that is, once the PDE has been solved on all component grids – in the hope that the resulting coefficients could approximate the full grid solution better than the classical combination. The following discussion is based on Kowitz's notation [KH14].

We illustrate the idea for an eigenvalue problem of the form

$$\mathcal{L}u = \lambda u, \tag{3.34}$$

where $\mathcal{L}$ is a linear differential operator (for example, the linear term in the gyrokinetic Vlasov equations), which can be discretized with a resolution level $\mathbf{i}$. We can then solve the eigenvalue problem on each of the component grids $\Omega_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{I}$.

As a next step, consider the functional

$$J(\vec{c},\lambda) = \left\|\mathcal{L}_{\mathbf{n}}\vec{u}^{(c)} - \lambda u_n^{(c)}\right\|_2 \quad \text{with} \quad u_n^{(c)} = \sum_{\mathbf{i}\in\mathcal{I}} c_{\mathbf{i}} u_{\mathbf{i}}, \tag{3.35}$$

where $\mathcal{L}_{\mathbf{n}}$ is the discrete full grid operator. One can then try to minimize J with respect to the eigenvalues λ and the coefficients $\vec{c}$, prolongating the component solutions $u_{\mathbf{i}}$ to the full grid $\Omega_{\mathbf{n}}$ with, for example, a multi-linear interpolation operator $P_{\mathbf{i}}^{\mathbf{n}}$. One can then show that minimizing (3.35) in a least squares sense is equivalent to solving the overdetermined eigenvalue problem

$$(\mathbf{L} - \lambda\mathbf{U})\vec{c} \approx 0, \tag{3.36}$$

where the matrices $\mathbf{L}$ and $\mathbf{U}$ are defined as

$$\mathbf{L} = [\mathcal{L}_{\mathbf{n}} P_{\mathbf{i}_1}^{\mathbf{n}} u_{\mathbf{i}_1} \quad \ldots \quad \mathcal{L}_{\mathbf{n}} P_{\mathbf{i}_N}^{\mathbf{n}} u_{\mathbf{i}_N}], \tag{3.37}$$

$$\mathbf{U} = [P_{\mathbf{i}_1}^{\mathbf{n}} u_{\mathbf{i}_1} \ldots P_{\mathbf{i}_1}^{\mathbf{n}} u_{\mathbf{i}_N}]. \tag{3.38}$$

Here, N is the number of component solutions, $N = |\mathcal{I}|$.

In the presence of faults, the OptiCom can simply exclude the failed solutions $u_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{J}$ and find the optimal combination coefficients in the reduced index set. This was done in [PHKH+15], but only with “offline” sequential tests (i.e., computing all component solutions without faults and then looking at the error of the combination solution assuming some of the solutions were missing).

* * *

We now turn our attention to our parallel implementation of the FTCT and the applications we tested it on. We have not yet implemented a parallel version of the OptiCom, but this would be an interesting topic for future research.

4 A Framework for the Combination Technique and Application Codes

In this chapter we want to introduce the software tools used throughout this thesis to carry out experiments with the combination technique. There are two components to consider. First, we need a framework to perform all steps specific to the combination technique: generating the index sets, setting up data structures for all component grids, hierarchizing and dehierarchizing the component grids and carrying out the combination step. The second component is the actual application being solved, which in our case should be a code to simulate a high-dimensional, time-dependent PDE (although of course we could also solve low-dimensional or stationary PDEs). Although the combination technique has been tested for over twenty years, only recently have there been efforts to develop a framework that implements the algorithm on a massively-parallel scale. This is one of the main objectives of our project *EXAHD*, and our partners at the IPVS Stuttgart have made significant contributions to reach this goal. The effort is motivated by the observation that the combination technique offers an additional level of parallelism on top of the application to be solved. This results from the fact that the component solutions $u_\mathbf{i}$ can be solved independently of each other, except for when combination is needed. It is nevertheless not obvious how to exploit this second level of parallelism optimally.

In this chapter we will describe the general parallelization strategy to solve the combination technique, and in the following chapters we will discuss how we have extended this framework to deal with both hard and soft faults. Additionally, this chapter will describe two application codes that have driven our experiments with the combination technique: the programming environment `DUNE`, and the plasma microturbulence code `GENE`. Both are highly optimized codes that are excellent benchmarks for the performance of the combination technique, but they are very different types of software. While `GENE` is a simulation code that solves a specific type of PDE, `DUNE` is a framework that simply provides data structures, methods and algorithms to solve PDEs more generally. By coupling the combination technique to both codes, we hope to show how flexible this approach is and encourage others to test the combination technique with other solvers out there, especially those with the potential to offer new physical insights. `GENE`, for instance, is of particular interest because plasma physicists need to reach higher levels of computational resolution in order to gain further insights into the behavior of plasma microturbulence. The combination technique offers a way to increase the resolution further without the need of faster supercomputers. This might prove crucial in the development of plasma fusion research.

4.1 Parallel Implementation of the Combination Technique

In the previous chapter we outlined the steps required by the combination technique. Now we are interested in determining which steps can be parallelized and how to parallelize them best. In Algorithm 2 we have sketched out what a parallel implementation of the combination technique could look like. Although it is relatively easy to come up with a simple parallel implementation of the combination technique, it is quite difficult to make sure the implementation scales on a massively parallel system. The major questions to answer are:

1. What is the best way to distribute the computational effort of solving the underlying PDE on a set of multiple component grids $\Omega_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{I}$? And should the chosen strategy depend on the number of grids, the dimension of the problem, or the number of unknowns in each grid?

2. What type of distributed data structures should we use for each component solution $u_{\mathbf{i}}$ and for the combined solution $u_n^{(c)}$?

3. What is the optimal communication strategy to combine the component solutions together?

Answering these questions has been the goal of our project partners at IPVS Stuttgart, and we refer to the PhD thesis of Mario Heene for detailed results.[1] For our purposes, it is enough to understand the overall parallelization strategy and discuss some of the main challenges faced and the ways to overcome them. In subsequent chapters we discuss how we extended this software framework to deal with both hard and soft faults.

A Manager-Worker Model for the Combination Technique

The starting point of a parallel implementation of the combination technique is to decide how to allocate the work of solving the PDE on $|\mathcal{I}|$ grids with a given pool of parallel processing elements. The parallelization strategy should be able to exploit the second level of parallelism of the combination technique, that is, the fact that component solutions can be computed independently of each other between combination steps. One way to achieve this is to divide all available processing elements into groups that perform work in parallel to each other. Say we have P available processes. We can divide them into G groups of equal size,

[1] To appear.

Algorithm 2: The Truncated Fault Tolerant Combination Technique in Parallel

```
input  : Parameter file settings.ini
output: Combined solution $u^{(c)}_{n,\boldsymbol{\tau}}$
1  Read in settings.ini ;                                                         // MP
2  Generate index set $\mathcal{I}^{d,n}_{\boldsymbol{\tau}} = \bigcup_{q=0}^{d+1} \mathcal{I}^{d,n}_{q,\boldsymbol{\tau}}$ and compute coefficients $c_{\mathbf{i}}$ ;   // MP
3  Create MPI groups and communicators ;                                          // MP
4  Distribute tasks among $G$ groups with index sets $\mathcal{I}_g$, $g = 0, \ldots, G-1$ ; // MP
5  for $g \in 0, \ldots, G-1$ do in parallel
6      for $\mathbf{i} \in \mathcal{I}_g$ do
7          $u_{\mathbf{i}} \leftarrow u(\vec{x}, t = 0)$ ;                         // Set initial conditions
8  while not converged do
9      for $g \in 0, \ldots, G-1$ do in parallel
10         for $\mathbf{i} \in \mathcal{I}_g$ do
11             $u_{\mathbf{i}} \leftarrow$ solve($u_{\mathbf{i}}$,$N_t$) ;          // Using domain decomposition
12         for $\mathbf{i} \in \mathcal{I}_g$ do
13             $u_{\mathbf{i}} \leftarrow$ hierarchize($u_{\mathbf{i}}$);
14     if faults detected then
15         recover() ;                               // Recover from faults: use the FTCT
16     for $g = 0, \ldots, G-1$ do in parallel
17         $u^{(g)}_{n,\boldsymbol{\tau}} \leftarrow \sum_{\mathbf{i} \in \mathcal{I}_g} c_{\mathbf{i}} u_{\mathbf{i}}$ ;   // Local combination
18     $u^{(c)}_{n,\boldsymbol{\tau}} \leftarrow \sum_{g=0}^{G-1} u^{(g)}_{n,\boldsymbol{\tau}}$ ;                        // Global combination
19     for $g \in 0, \ldots, G-1$ do in parallel
20         for $\mathbf{i} \in \mathcal{I}_g$ do
21             $u_{\mathbf{i}} \leftarrow$ sample($u^{(c)}_{n,\boldsymbol{\tau}}$) ;   // Extract $u_{\mathbf{i}}$ from combination solution
22             $u_{\mathbf{i}} \leftarrow$ dehierarchize($u^{(c)}_{n,\boldsymbol{\tau}}$) ;   // Transform back to nodal basis
```

each thus having P/G processes. Each group can then be assigned a subset of all the of component grids $\Omega_{\mathbf{i}}$ on which the PDE has to be solved. Recall that there are $|\mathcal{I}| = \mathcal{O}(d(\log h_n^{-1})^{d-1}) = \mathcal{O}(dn^{d-1})$ such component grids, so each group receives a subset $\mathcal{I}_g \subset \mathcal{I}$, $g = 0, \ldots, G-1$, of the total index set, thus being assigned $|\mathcal{I}_g| = \mathcal{O}(dn^{d-1}/G)$ component grids.

Additionally, we define one extra process that will coordinate the different steps of the combination technique. We call this the *manager process*. Its tasks include generating the set of combination indices $\mathcal{I}$ (line 2), define the MPI groups and communicators (line 3), distributing the workload among the process groups (line 4) and coordinating the combination steps (lines 17-18). Finally, each process group defines a *master process* that communicates with the manager process. This means there are $G+1$ MPI communicators: one for each of the G groups for intra-

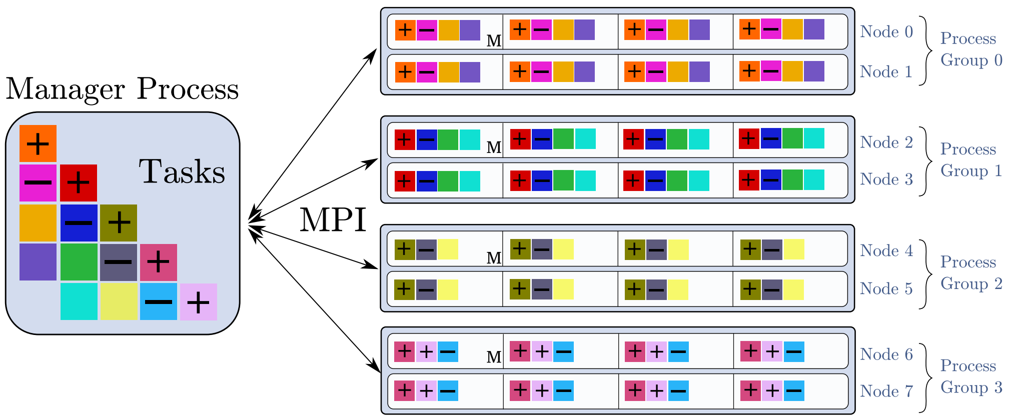


Figure 4.1: Manager-worker model for the combination technique. In this example, 14 tasks are distributed among $G = 4$ process groups, each composed of two nodes, with four processes per node.

group communication, and an additional global commmunicator involving all G master processes and the manager process for inter-group communication.

The four parallel for-loops in Algorithm 2 exploit the second level of parallelism, since the groups work independently of each other. But even though tasks are processed sequentially within a group, each step is parallelized at the first level. This includes the `solve`, `hierarchize`, `combine`, `sample` and `dehierarchize` steps. They are all performed in parallel. The details of how this is done can be found in [HP16b].

This parallelization scheme could look something like the diagram depicted in Fig. 4.1. We depict a two-dimensional fault-tolerant combination technique that results in a set of 14 component grids on which the PDE has to be solved. We use the term *task* to refer to each of the individual problems that result from the combination technique. A task is therefore an object that contains all relevant information about a component solution: its level vector $\mathbf{i}$, its combination coefficient $c_{\mathbf{i}}$, the dimension of the problem, the boundary conditions, the PDE to be solved, the number of time steps to perform, the time step width, etc.

In this sample architecture we assume we have eight computing nodes, each with four processes. Here we chose to define a process group to be composed of eight processes, so each group encompasses two nodes. The four groups define a master process (denoted with an M) which communicate via MPI with the manager process. Once the tasks are distributed evenly among the groups, the manager process triggers the start of the computation step, and each group proceeds to solve the set of tasks assigned to it, one task after another. Notice that each task is distributed among the processes in the group. This represents the domain decomposition of each task, so in this example, each task is divided into eight partitions, and each process solves its part of the domain.

It is not straightforward to define the number of groups to be used. This decision requires finding a balance between the two levels of parallelism. On the first level, we want each individual task to be solved fast, which would mean using many processes per task (a fine-grained domain decomposition). But this would result in fewer groups, which reduces the parallelism on the second level. In practice, we usually first determine a reasonable domain decomposition per task (say, eight processes per task). This automatically defines the number of groups we can use.

The algorithm reads in a parameters file (`settings.ini`) that contains the basic information about the simulation and it uses the basic parser language structure. For the example depicted in Fig. 4.1, it would contain at least the following information:

```
[ct]
lmin = 1 1
lmax = 5 5
p = 4 2
ncombi = 10
[application]
dt = 1e-3
nsteps = 100
[manager]
ngroup = 4
nprocs = 8
```

The parameters correspond to the following quantities:

- $\mathbf{l}_{\text{min}}$ is related to the truncation parameter $\boldsymbol{\tau}$ through the relation $\mathbf{l}_{\text{min}} = \mathbf{1} + \boldsymbol{\tau}$.
- $\mathbf{l}_{\text{max}}$ is the full grid resolution $\mathbf{n}'$ we're trying to approximate, $\mathbf{l}_{\text{max}} = \mathbf{n}' = \mathbf{n} + \boldsymbol{\tau}$.
- $\mathbf{p}$ corresponds to the parallelization vector for the domain decomposition. In this example, $\mathbf{p} = 4\ 2$ means "parallelize the domain using a partition of 4 sub-domains in x and 2 sub-domains in y".
- `ncombi` is the number of times we combine the component solutions throughout the simulation.
- dt is the simulation time step.
- `nsteps` is the number of time steps to perform between each combination step.
- `ngroup` and `nprocs` are the number of process groups and the number of processes per group, respectively. We should have $\prod_{i=0}^{d-1} p_i =$ `nprocs`.

In order for this parallelization strategy to scale, the individual components should scale, and an appropriate load balancing scheme has to be used. The steps that should be made to scale are `hierarchize`/`dehierarchize`, `combine` and `sample`, since they require local (within groups) and global (across groups) communication.

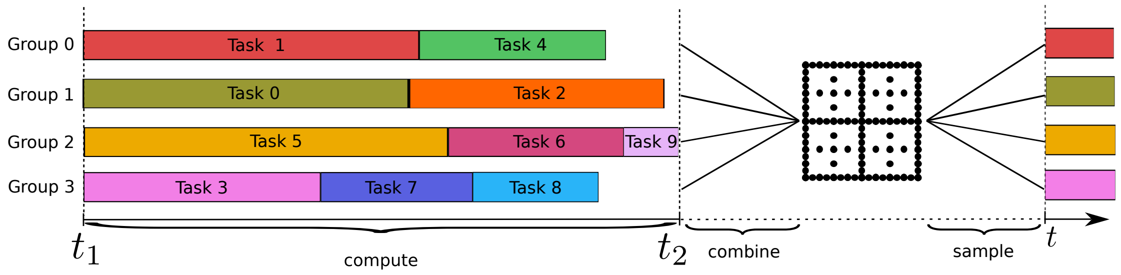


Figure 4.2: Load balancing, combination and scattering steps using the manager-worker model.

Communication

Hupp has extensively studied efficient algorithms to hierarchize and dehierarchize component solutions [Hup13,Hup14,HJ]. Based on his results, Heene implemented a highly efficient scheme to perform these steps on a distributed system, optimizing it to fit the manager-worker strategy [HP16a]. The resulting implementation scales perfectly in experiments involving as many as 32k cores.

Further effort has been invested into making sure that the combination step scales. Hupp et al. have developed highly efficient, parallel algorithms that exploit the hierarchical structure of sparse grids in order to minimize the communication cost of combining a set of component solutions into a sparse grid [HHJP16]. These investigations served as the basis to develop a highly scalable combination algorithm for distributed systems by Heene and Pflüger [HP16b]. Their implementation scaled well in experiments with up to 180k cores.

Load Balancing

One last problem involves determining how to distribute the tasks among the process groups such that the workload is balanced. Heene et al. observed that the time to solve N time steps on a given component grid $\Omega_{\mathbf{i}}$ depends not only on the number of grid points, but also on the grid's degree of anisotropy [HKP13]. Based on this observation, they developed a model to estimate the time needed to solve each component solution $u_{\mathbf{i}}$, which gives a way to distribute the workload evenly among the process groups.

Figure 4.2 illustrates a typical workflow for the `solve`, `combine` and `sample` steps (the `hierarchization` and `dehierarchization` steps have been left out for simplicity). We see ten tasks distributed among four groups, and each group solves its set of tasks sequentially. Since the workload is balanced, the four groups should take similar times to perform the `solve` step on all grids. The results are then combined into the sparse grid, and the combined solution is used as starting point for the next set of time steps.

Related Work

Although the combination technique has been applied to a wide range of applications, most of the experiments so far have been carried out sequentially (i.e., not exploiting the parallelism across component solutions). One early study of the combination technique in parallel can be found in [Gri92] and [GHZ96]. One more recent effort to parallelize the combination technique has been described by Strazdins, Ali and Harding [SAH15]. Their approach differs from ours in that it does not divide the component grids among process groups, which affects the way solutions are combined. They showed that their implementation scales to at least 1,500 cores and up to 3,000 cores for 2D and 3D experiments. It is not clear, however, whether their approach can scale on massively parallel systems or how well it performs for higher dimensions.

4.2 Application Codes

Having a general framework for the combination technique in place, we can couple it with different application codes to be called during the `solve` step. The only requirements are 1) that we can access the underlying solution field $u_{\mathbf{i}}$, 2) that the code is discretized on a full Cartesian grid $\Omega_{\mathbf{i}}$ for which $\mathbf{i}$ can be specified by the user and 3) that the time step size can also be varied. In this section we introduce two codes that we have used to test the combination technique: `GENE`, a plasma simulation code, and `DUNE`, a framework to solve a wide range range of PDEs.

4.2.1 Plasma Simulations with `GENE`

One of the main motivations behind this thesis is the prospect of investigating physical scenarios that are of real interest to the scientific community. As we mentioned in the introduction of this thesis, one such scenario is the fusion process that takes place in highly magnetized hot plasma, which is exactly what happens in the sun's core. Physicists are interested in approximating these conditions in reactor devices by fusing tritium and deuterium atoms. In order to achieve this, the plasma has to be confined and heated up to a temperature of roughly 100 million degrees Kelvin. In the sun, the plasma is confined purely by the star's gravitational pull. Since this cannot be replicated in the laboratory, the plasma is instead confined using a strong magnetic field, which can be optimized by using special reactor geometries [Mer09]. In Fig. 4.3 we show the two most common geometries used for magnetic plasma confinement. Tokamak devices are torus shaped and are currently the most common type of confinement devices. The reactor that is being built by the international joint experiment ITER, which aims to show the feasibility of plasma fusion as a sustainable source of energy, is based on the tokamak geometry [ite].

The development of such complex devices is necessarily accompanied by large scale numerical simulations. It is crucial to try to understand the anomalous transport and microturbulent phenomena that characterize the plasma flow, since

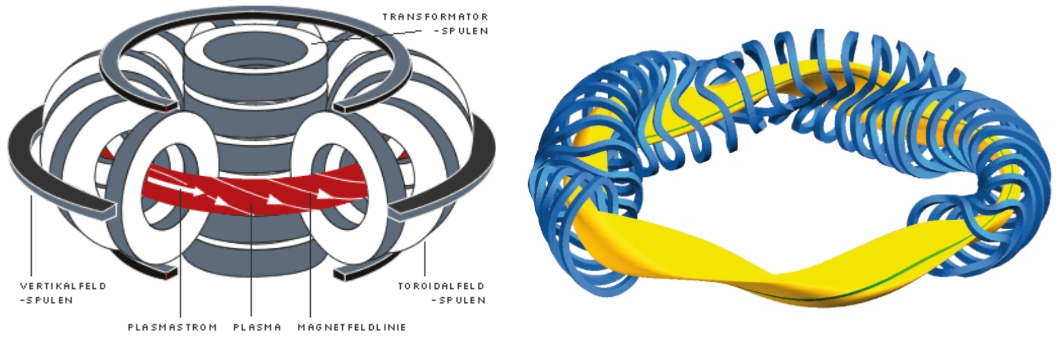


Figure 4.3: The two most common plasma device geometries: the tokamak (left) and the stellerator (right). Source: IPP

it leads to a deterioration of the process of plasma fusion [Mer09]. The equations that govern the evolution of a magnetically confined plasma field are challenging to solve, since the distribution function of the field is, in the so-called *gyrokinetic approximation*, a five-dimensional function u_s given by the following set of nonlinear partial integro-differential equations [Kow16]:

$$\frac{\partial u_s}{\partial t} + \left(v_\parallel \frac{\vec{B}_0}{|\vec{B}_0|} + \frac{B_0}{B^*_{0\parallel}} \vec{v}_{\text{drift}} \right) \cdot \left(\nabla u_s + \frac{1}{m_s v_\parallel} \left(q\bar{\vec{E}}_1 - \mu \nabla (B_0 + \bar{B}_{1\parallel}) \right) \frac{\partial u_s}{\partial v_\parallel} \right) = 0. \tag{4.1}$$

Here, $\vec{v}_{\text{drift}}$ includes the different drift velocities of the charged particles. Both magnetic and electric fields $\vec{B}$ and $\vec{E}$ are separated into a background Maxwellian distribution and a perturbation term, $\vec{B} = \vec{B}_0 + \vec{B}_1$ and $\vec{E} = \vec{E}_0 + \vec{E}_1$. The subscript $\parallel$ refers to the direction parallel to the magnetic field, and the fields denoted with bars are the *gyro-averaged* fields[2].

For numerical purposes, the coordinate system has to be aligned to the magnetic field, since the interesting microturbulence phenomena are highly anisotropic. After such a transformation, the solution u_s depends on five quantities: x, y, z, $v_\parallel$ and μ. x and y are also called *radial* and *binormal* directions. z is called the parallel direction, and it follows the magnetic field line. $v_\parallel$ is the velocity in this same direction, and μ is the magnetic moment. The subscript s in u_s refers to the *species* under consideration, which are usually either ions or electrons. It is typical to simulate $1-4$ species. If we put together the different species into one vector $u \equiv u(x, y, z, v_\parallel, \mu, s)$, the set of equations (4.1) have the general form

$$\frac{\partial u}{\partial t} = \mathcal{L}(u) + \mathcal{N}(u)\,, \tag{4.2}$$

where $\mathcal{L}$ and $\mathcal{N}$ represent the linear and nonlinear parts of the integro-differential operator.

[2]The charged particles rotate with high velocity around the magnetic field lines, and the gyrokinetic model as presented here is the result of averaging out this velocity component. For more details, see [Mer09].

The code `GENE` [J+00] solves system (4.2) for different scenarios. It is common to investigate only the linear operator, either by solving the time-dependent system or by computing the eigenvalues of $\mathcal{L}$, which allows one to study the mechanisms that drive the microinstabilities and to predict certain properties of the nonlinear regime. `GENE` uses full Cartesian grids $\Omega_{\mathbf{i}}$ to discretize the five-dimensional domain, with 2^{i_j} grid points in each dimension $j = 1, \ldots, d$.

Given the large grid sizes, it is common to simulate only certain sections of the plasma field, in what is called the *local* approximation. In this mode, only flux-tubes with a small radial extent are simulated, under the assumption that the scale of the turbulent effects is much smaller than the characteristic gradient scales in the radial direction [NTJ+16]. This approximation allows one to use periodic boundary conditions in the x and y directions, so one can substitute them by their Fourier transforms k_x and k_y, which reduces the computational effort considerably. The z and $v_{\|}$ directions are discretized with an Arakawa scheme of order two. To integrate in the μ direction, `GENE` implements Gauss and trapezoidal rules. For the time integration, `GENE` uses a fourth order Runge-Kutta scheme. The code is written in Fortran and parallelized with both MPI and OpenMP.

Although local simulations provide valuable insights into the behavior of the plasma, there are various phenomena that can only be well understood with *global* simulations, where one simulates plasma surfaces over an extended radius. These simulations are much more expensive. In idealized conditions, local and global simulations agree, but for more realistic scenarios, global simulations are needed. They seem to be necessary to understand specific phenomena that arise in real confinement devices, such as ASDEX Upgrade and JET [AGJT15]. Currently, the largest simulations have to be run on parallel systems with tens and up to hundreds of thousands of cores. Two noteworthy parallel experiments include a global simulation requiring up to 64,000 cores [GLB+11] and a local simulation using 262,000 cores [MF10]. Scientists hope that future generation supercomputers will allow more detailed scenarios to be investigated, and new numerical techniques could facilitate this process.

4.2.2 The `DUNE` Framework

One of the most attractive characteristics of the combination technique is that one can solve a wide range of problems by changing the underlying PDE solver. `GENE` is one example, but the combination technique has been used for problems ranging from the Schrödinger equation [GG00] to turbulent flow problems using the Navier-Stokes equations [GHZ96]. Each code defines a discretization scheme (finite differences, finite elements, finite volumes, etc.), and this scheme defines the type of grid used. This makes most codes hard to extend, since they focus on one specific data structure. This is the motivation behind `DUNE` (or *Distributed and Unified Numerics Environment*), a framework developed by several research groups (mostly based in Germany) whose main goal is to offer abstract data structures with which a very general class of grid-based solvers can be implemented [BBD+08].

DUNE is written using C++ templates, static polymorphism, traits, and other modern programming techniques in order to optimize the runtime performance of the code. Its design is based on three core principles: *flexibility* (users can add new components), *efficiency* (maximize computational performance) and *legacy code* (it should be possible for users to include existing applications into their new libraries). Additionally, DUNE is based on the idea that data structures and algorithms should be separated, which makes codes easier to maintain and extend.

The code is organized into modules, each of which implements specialized tasks. As of version 2.4, the core modules are [BBD$^+$16]

- `dune-grid`: defines the generic interface for a grid and contain several specific implementations of grids (e.g. simplicial grids, hexahedral and tetrahedral grids, etc.). It implements a very general definition of grids, defined in [B$^+$08] (nonconforming, hierarchically nested, multi-element-type, parallel grids in arbitrary dimensions).
- `dune-geometry`: implements different types of reference elements, their mappings and quadratures.
- `dune-grid-howto`: tutorial with basic functionalities.
- `dune-localfunctions`: a library of functions on the reference elements.
- `dune-common`: includes all common functionalities across the modules.
- `dune-istl` – *Iterative Solver Template Library*: defines classes for sparse vectors and matrices, including data structures and solvers.

DUNE has been highly optimized and offers many parallel capabilities. This makes it an attractive application code for the combination technique.

In this thesis we worked with the additional module `dune-pdelab`, which allows one to implement prototypes for a wide range of solvers quickly, including many elliptic, parabolic and hyperbolic PDEs, discontinuous Galerkin FEM, incompressible NS equations, Maxwell's equations, among others [BHM10].

* * *

In the rest of the thesis we will see how to extend the software framework introduced in this chapter to make it tolerant to both hard and soft faults. We will also discuss how one can couple both GENE and DUNE to the framework and how the combination technique performs with these two application codes.

5 Dealing with Hard Faults with the Combination Technique

At this point we have all the necessary ingredients to test the combination technique in the presence of hard faults. In Chapter 2 we discussed what hard faults are and how they affect HPC systems. In Chapter 3 we introduced the fault tolerant combination technique (FTCT), a PDE solver that can recover when part of the data is missing, independently of what caused the data to go missing. Finally, in Chapter 4 we described one possible way to parallelize the combination technique in order for it to run efficiently on large parallel systems. Now we want to answer several questions regarding the performance of the fault tolerant combination technique, in particular:

- How good is the combination solution after it has been affected by faults?
- How much does it cost to ensure fault tolerance compared to checkpoint/restart?
- How well does the FTCT scale?

We will show two sets of experiments to address these questions. We will first look at some preliminary tests in serial to get a feel for the approximation quality of the FTCT when simulated faults occur. We do this using the code `GENE`, injecting multiple hard faults and observing the quality of the combined solution as well as the overhead involved in recovering compared to recomputing the solutions straightforwardly. Afterwards we will move on to large scale parallel experiments using `DUNE` and confirm the results obtained in the serial experiments.[1] These experiments will also help us investigate the parallel overhead of the FTCT and its performance on a large parallel system. The results of this second set of experiments are meant to prove that our implementation of the FTCT could be suitable for future exascale systems.

[1]The reason we chose `DUNE` instead of `GENE` for our first parallel experiments is that `GENE` requires additional, non-trivial adjustments in order for it to run properly with the combination technique. These adjustments were not implemented by the time the parallel framework was ready to run, whereas `DUNE` was already available for testing.

5.1 Preliminary Studies with GENE

Before implementing the FTCT efficiently and in parallel we set off to test its performance with a series of experiments in serial. To do this we chose the code `GENE` , since we had access to an implementation of the combination technique developed by Kowitz [KPJH12]. The code is a Python interface that communicates with `GENE` and offers the following core functionalities (as Python classes):

- `CombinationSchemeArbitrary`: Generates the set $\mathcal{I}_{\boldsymbol{\tau}}^{d,n}$ of combination levels for a given n and $\boldsymbol{\tau}$, along with the corresponding combination coefficients $c_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{I}_{\boldsymbol{\tau}}^{d,n}$.
- `geneEnvironment`: Defines the system path where `GENE` is located, the path with the parameter file and the path where the `GENE` solutions should be stored after the simulations complete.
- `DataProviderGene`: Runs the `GENE` simulations in a given `geneEnvironment`, acting as a communicator between the Fortran code and the Python classes.
- `geneCombinationGridIV`: Provides the multidimensional data structures to store the `GENE` solution fields that result from the initial value simulations.
- `combineGrids`: Encapsulates the algorithms to combine the grids defined by the `CombinationSchemeArbitrary` and to compute the combined solution.

In order to include the fault tolerant combination technique, we implemented the class `CombinationSchemeFaultTolerant`, which inherits from the `CombinationSchemeArbitrary` class. This class has two main additional functionalities, namely:

1. It generates the extended index set $\bigcup_{q=0}^{d+1} \mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$, and
2. It solves the GCP (Eq. 3.32) to find new combination levels and coefficients to exclude the missing combination solutions $u_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{J}$.

Our implementation for the serial experiments is summarized in Algorithm 3. We first compute a reference solution of level $\mathbf{n}$ which is used to calculate the error of the consequent combinations. Then `GENE` is called on every component grid (line 8) and the fault-free combination is performed (line 12). We then choose M random component solutions to fail and solve the GCP to find alternative combination coefficients that exclude the failed solutions. Notice that this fault simulation is done “offline”: we remove component solutions *after* the the simulation finishes instead of injecting faults in real time, since we do not have any fault detection mechanisms in place at this stage, but this is not necessary for the types of measurements we are interested in. Finally, for each of the combination solutions (without faults, with faults and recovered) we compute the error with respect to the reference solution. Although it is not specified in the algorithm, we also measure the computation time needed at each call of `runGene`.

Algorithm 3: Serial Fault Tolerant Combination Technique with GENE

```
input  : Paths to GENE binaries; sparse grid resolution $n$; truncation
         parameter $\boldsymbol{\tau}$; total time steps $N_t$; number of faults to simulate
         $M$
output: Combined solution $u^{(c)}_{n,\boldsymbol{\tau}}$
1  scheme = CombinationSchemeFaultTolerant($n, \boldsymbol{\tau}$) ;   // Generate extended
     index set and $c_{\mathbf{i}}$
2  gene_env = geneEnvironment(genePaths) ;   // Define a geneEnvironment
   // GENE reference solution
3  provider_ref = DataProviderGene(gene_env, $\mathbf{n}$)
     provider_ref.runGene($N_t$);
4  grid_ref = geneCombinationGridIV(provider_ref);
5  $u_{\mathbf{n}}$ = grid_ref.data;
   // Combination technique
6  combi_grid = combineGrids(scheme);
7  for $\boldsymbol{i} \in \bigcup_{q=0}^{d+1} \mathcal{I}^{d,n}_{q,\boldsymbol{\tau}}$ do
8      provider = DataProviderGene(gene_env, $\mathbf{i}$) ;   // Providers for GENE
9      provider.runGene($N_t$) ;   // Run $N_t$ steps of GENE on current level
10     grid = geneCombinationGridIV(provider);
11     combi_grid.addGrid(grid)
12 $u^{(c)}_{n,\boldsymbol{\tau}}$ = combi_grid.getCombination() ;   // Combination solution without faults
13 $e_{\mathrm{CT}}$ = L2Error($u^{(c)}_{n,\boldsymbol{\tau}}, u_{\mathbf{n}}$) ;   // Error of combination technique
   // Simulate faults
14 $\mathcal{J}$ = scheme.generateRandomFaults($M$);
15 $u_{\mathrm{Faults}}$ = combi_grid.getCombination() ;   // Combination with faults
16 $e_{\mathrm{Faults}}$ = L2Error($u_{\mathrm{Faults}}, u_{\mathbf{n}}$) ;   // Error of combination with faults
17 scheme.recover($\mathcal{J}$) ;   // Solve GCP to find new combination technique
18 $\tilde{u}^{(c)}_{n,\boldsymbol{\tau}}$ = combi_grid.getCombination() ;   // Combine with new coefficients
19 $e_{\mathrm{FTCT}}$ = L2Error($\tilde{u}^{(c)}_{n,\boldsymbol{\tau}}, u_{\mathbf{n}}$) ;   // Error with new combination coefficients
```

5.1.1 Simulation Scenario

To test our implementation we used a standard test case in GENE that simulates a trapped electron mode (TEM) and a mode driven by the ion temperature gradient (ITG). In this scenario we simulate two species – electrons and deuterium ions – using a realistic mass ratio, which causes the space and time scales to decouple strongly. In this scenario there are two exponentially growing modes (meaning the linear operator has two eigenvalues with positive real part) [Kow16]. We use a standard tokamak geometry for the fusion device. This is a common test case in gyrokinetics that is nevertheless realistic. The GENE parameter file for this experiment (referred to as kinetic small) can be found at the end of this chapter.

We perform two sets of experiments. The first one is meant to demonstrate that the fault tolerant combination technique approximates the full grid solution well after faults occur. We choose a reference solution of moderate size in order to be able to compute the error. For this set of experiments we perform a three-dimensional combination in the $(\mu, v_{\|}, z)$ dimensions. The other two dimensions (k_x and k_y) are kept fixed. In the second set of experiments we perform a four-dimensional combination in $(\mu, v_{\|}, z, k_x)$. We also choose a higher reference level $\mathbf{n}$ but do not compute the reference solution. The aim of this set of experiments is to measure the overhead involved in recovering from faults, including the computation of the additional component grids to ensure fault tolerance.

Although we perform the combination technique in serial, each instance of `GENE` can still be computed in parallel. This level of parallelism is part of `GENE` and therefore does not require any additional changes in our implementation. We performed our experiments on four nodes of the MAC cluster *Cloaca*[2], each of which has 8 cores. Each node is equipped with two Intel Xeon E5-2670 (SandyBridge-EQ), 128 GB RAM and a QDR infiniband connection.

5.1.2 Results

Convergence

We chose a reference solution of level $\mathbf{n} = (5, 7, 6)$ in dimensions $(\mu, v_{\|}, z)$. For the spatial dimensions k_x and k_y we use five and one discretization points respectively. The truncation parameter $\boldsymbol{\tau}$ for these experiments was set to $\boldsymbol{\tau} = (1, 3, 2)$. The truncated combination technique that results from this choice of parameters has 19 component grids. The extended index set to ensure fault tolerance $\bigcup_{q=0}^{d+1} \mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$ adds only one more component grid for a total of 20. Grouped by their resolution level we have 10, 6, 3 and 1 component grids for $q = 0, 1, 2, 3$ respectively.

We randomly simulated 1 to 5 faults, which means that 1 to 5 out of the 20 component solutions fail. Additionally we carried out three subsets of simulations, combining once after either 100, 1,000 or 10,000 time steps. In all cases we used time step size of 2×10^{-4} s. At the end of each simulation we computed the relative l_2 error given by

$$e = \frac{\|u_{n,\boldsymbol{\tau}}^{(c)} - u_{\mathbf{n}}\|_2}{\|u_{\mathbf{n}}\|_2} \tag{5.1}$$

with respect to the full grid solution.

For each of the 15 sets of experiments we performed 50 simulations in order to obtain more reliable statistics, choosing at each run new random solutions to fail. Figure 5.1 summarizes our error measurements, showing the average over the 50 simulations. There are several interesting observations. First, we see that, for this example, combining after a small number of time steps (100) results in a smaller error than after a larger number of time steps (10,000). We also notice that the error of the combination technique after recovering from faults is very close to that of the combination without faults. This error becomes larger as the number of

[2]`http://www.mac.tum.de/wiki/index.php/MAC_Cluster`

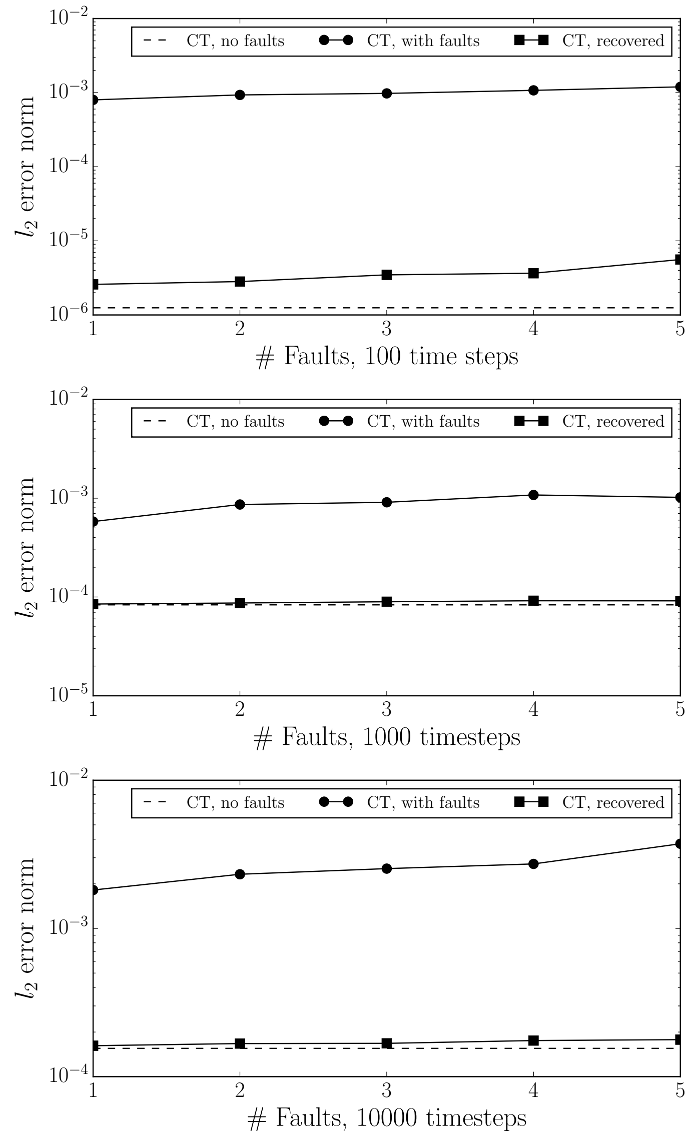
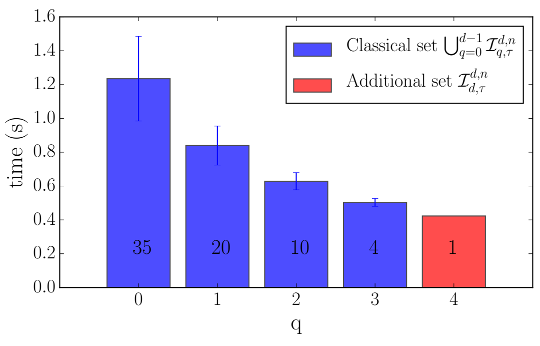
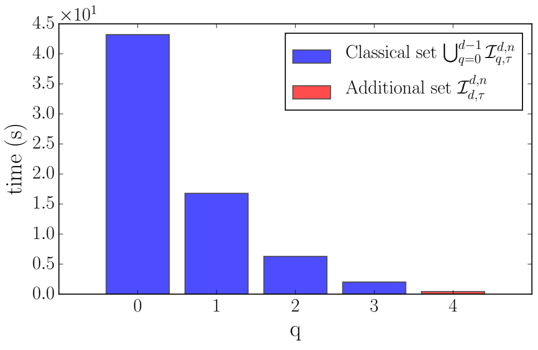


Figure 5.1: Approximation error of the combination technique applied to GENE without faults (*dashed*), with faults (*circles*), and after recovery (*squares*). On the x axis we show the number of component solutions that failed, and the error was measured after 100, 1000 and 10000 time steps.

faults increases but even for a high percentage of faults the error remains very close to that of the fault-free combination. Notice that the highest number of faults in our experiments corresponds to a failure rate of 25% (5/20), which we believe is a remarkable result. We also plot the error of the combination technique with faults, obtained by combining the component solutions that did not fail with their usual combination coefficients. This results in a loss of one to two orders of magnitude in the approximation quality as measured by this error norm. For additional visual proof we plot two different slices of the five-dimensional distribution function of the plasma in Fig. 5.4. This answers our first question at the beginning of the



(a)



(b)

Figure 5.2: **(a)** Computation times required by the different component solutions in GENE grouped according to their resolution level q. We show the mean and standard deviation of the 50 runs with 1,000 time steps. The number in each bar indicates the number of grids for each level q. **(b)** Total accumulated computation times of the component grids.

chapter: the approximation quality of the combination solution after recovering from faults is very good.

At this point we would like to address a valid question that arises in discussions around the convergence of the FTCT: if the error is almost as good as without faults, why don't we use fewer component grids from the beginning? As we argued in Chapter 3, the original combination technique without faults (which results in the usual sparse grids as we know them) is already an optimal trade-off between precision and computational cost (at least a priori), so any further benefit in cost (i.e., having fewer grids) results in an increase of the error that does not make the trade-off worth it. We are willing to incur this cost to be fault tolerant, but our target solution should be the original sparse grid. Furthermore, if we start with a smaller combination technique with fewer grids and then allow faults to occur, the error of the recovered combination would be even larger, which is not something we want.

Costs

To answer the second question (how much it costs to ensure fault tolerance using the GCP as opposed to checkpointing) we measured two things: 1) The additional effort required to compute extra component solutions (namely, those in $\bigcup_{q=d}^{d+1}\mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$) and 2) the time it takes to recompute any failed solutions that are needed for the alternative combination technique (those with $q \geq 2$ that fail *and* have a nonzero coefficient after the GCP is solved). To address these points we use a much larger combination technique in four dimensions $(\mu, v_{\|}, z, x)$ (still using only one point in the k_y dimension) with $\boldsymbol{\tau} = (1, 3, 2, 1)$ and $\mathbf{n} = (6, 8, 7, 6)$. This choice results in 70 component solutions partitioned as follows:

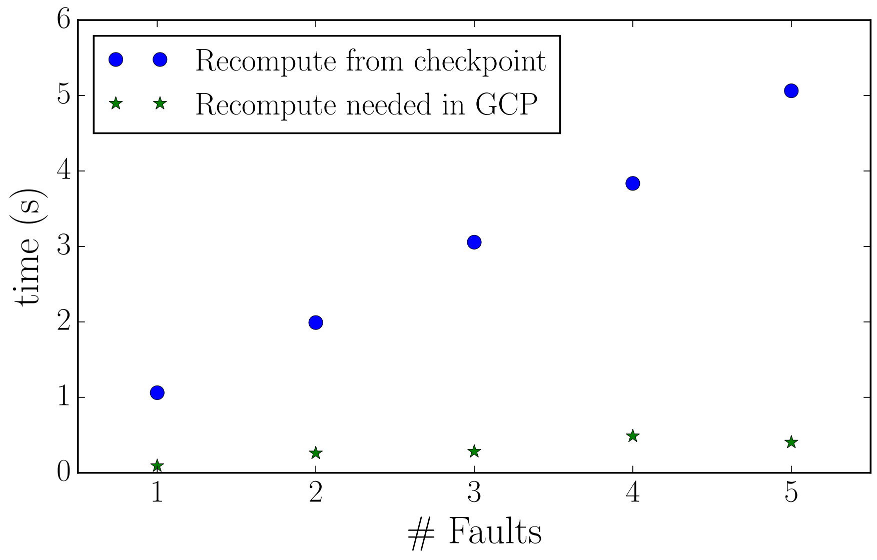


Figure 5.3: The accumulated time to recompute all lost component solutions from its initial state grows roughly linearly with the number of faults. The fault tolerant combination technique, on the contrary, requires recomputation only in some cases and for very coarse component solutions ($q \geq 2$).

$$|\mathcal{I}_{0,\boldsymbol{\tau}}^{4,15}| = 35, \quad |\mathcal{I}_{1,\boldsymbol{\tau}}^{4,15}| = 20, \quad |\mathcal{I}_{2,\boldsymbol{\tau}}^{4,15}| = 10, \quad |\mathcal{I}_{3,\boldsymbol{\tau}}^{4,15}| = 4, \quad |\mathcal{I}_{4,\boldsymbol{\tau}}^{4,15}| = 1. \tag{5.2}$$

Once again, only one additional component solution is needed in this case to ensure fault tolerance. Figure 5.2a shows the average time that a component solution of resolution q needs to perform 1,000 time steps. We label each bar with the number of component solutions corresponding to a given level q. The time decreases as the grid resolution decreases (higher q) although not linearly as we would expect. Figure 5.2b shows the accumulated time for the corresponding component grids. It becomes clear that the computational effort required by the additional grids (red bar) is very small – in this experiment it represents only 0.6% of the total computation time. The result from Proposition 3.3.1 establishes that for this case we should expect costs from the FTCT of at most 2.4% in terms of additional grid points, so the orders of magnitude seem to agree. The time it takes to solve the GCP itself is negligible and has therefore not been included in the total costs.

Finally, we are interested in the computational effort required to recompute some of the coarse component solutions when applying the fault tolerant combination technique as opposed to recomputing all failed solutions from the last checkpoint. Figure 5.3 shows these measurements, where we have plotted the average accumulated time needed to recompute the failed solutions (again for 1,000 time steps) in both cases. Whereas the time grows linearly when recomputing all failed solutions,the cost fault tolerant combination technique remains small even for a large number of faults. In many cases, no recomputation is needed at all.

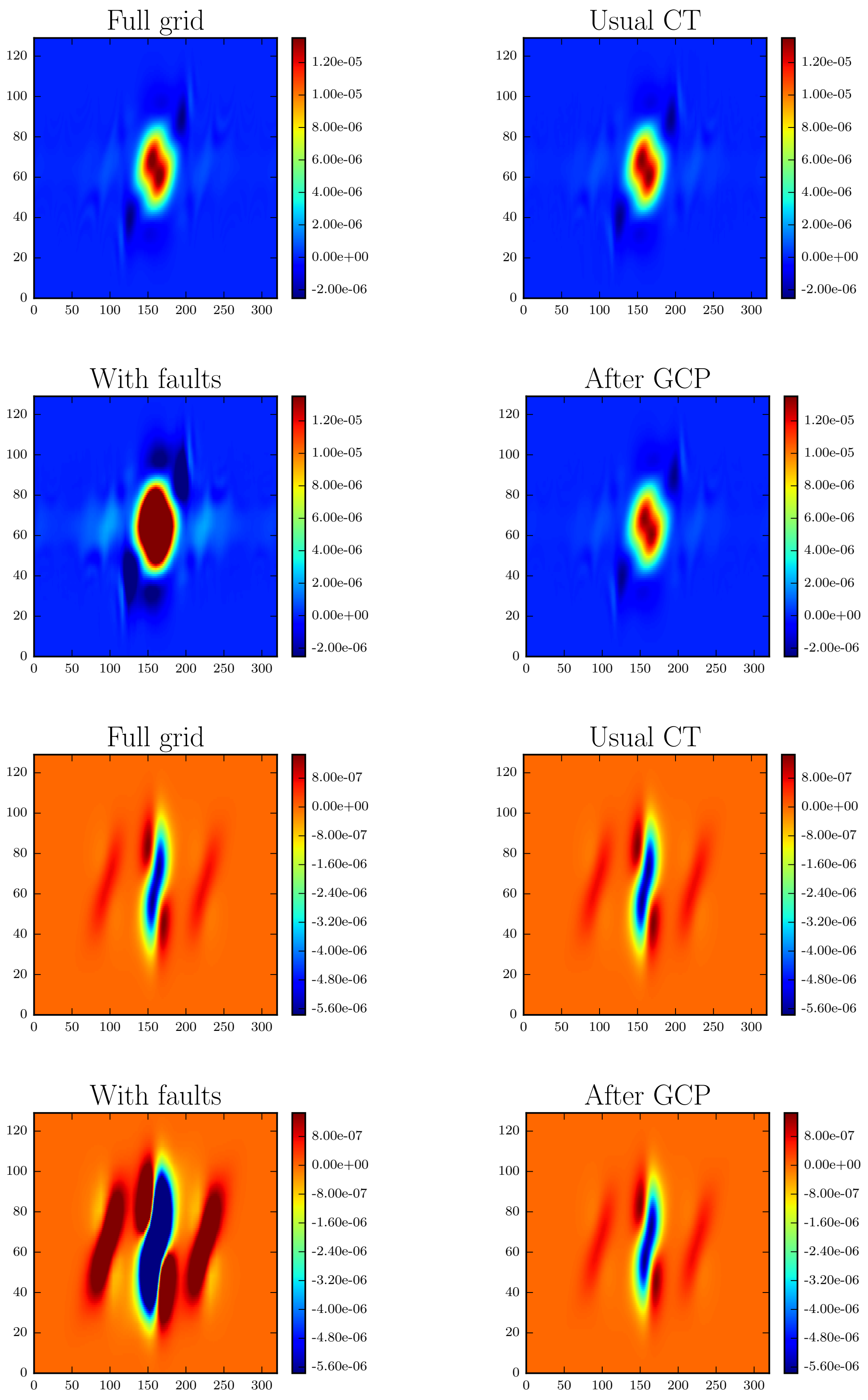


Figure 5.4: Two different $zx - v_{\|}$ slices of the GENE distribution function after 10,000 time steps. *CT* refers to the usual combination technique without faults and *After GCP* refers to the combined solution after finding new combination coefficients by solving the GCP.

Appendix: GENE parameter file

```
&box
kymin =    0.3
lv     =    3.00
lw     =    9.00
lx =    4.16667
adapt_lx = T
mu_grid_type =
          'clenshaw_curtis'
/
&general
nonlinear =    F
arakawa_zv =    T
arakawa_zv_order =    2
calc_dt = F
dt_max        =    7.39E-4
courant       =       1.0
beta          =    0.1E-02
debye2        =       0.0
collision_op = 'none'
init_cond = 'fb'
hyp_z =      2.000
hyp_v =    0.5000
/
```

```
&geometry
magn_geometry = 'circular'
q0          =      1.4
shat        =    0.8
trpeps      =    0.18
major_R     =      1.0
major_R     =      1.0
norm_flux_projection   = F
/
&species
name    = 'ions'
omn     =      2.0
omt     =      4.5
mass    =      1.0
temp    =      1.0
dens    =      1.0
charge =  1
/
&species
name    = 'electrons'
omn     =      2.0
omt     =      3.5
mass    =    0.27E-03
temp    =      1.5
dens    =      1.0
charge = -1
/
```

5.2 Large-Scale Experiments in Parallel

Motivated by the results of our serial experiments, we decided to extend our parallel C++ framework from Chapter 4 to be able to recover from hard faults. Several research groups have tried to address similar problems from different perspectives and as we will see we can benefit from some of their insights. Most noticeably, the *Fault Tolerance Working Group*, an independent group of researchers in the MPI community, has developed the *User Level Failure Mitigation Specification* [B$^+$12], an MPI implementation that allows the user to manage failures in an application. Their work addresses the kind of problems that we are interested in, so it is worth briefly describing some of its main functionalities.

5.2.1 The User Level Failure Mitigation Specification (ULFM)

As any regular user of MPI can attest, the way it deals with faults is to abort the whole MPI network, returning the error handler `MPI_ERRORS_ARE_FATAL`. Sometimes it is possible to use the alternative error handler `MPI_ERRORS_RETURN`), after which the network will not shut down but the state of the application will be undefined [Wal16]. ULFM is an effort to define a precise behavior that MPI should exhibit in the presence of faults, since the MPI standard does not offer this. Its design has the following four goals in mind [BBH+13]:

1. Maximize flexibility, so that the user can define how to respond to faults depending on the application at hand. MPI should therefore not try to repair the application itself.

2. Ensure that no MPI operations stall when failures occur.

3. Ability to be implemented within legacy codes.

4. Keep the overhead as low as possible outside recovery periods.

Figure 5.5 outlines the basic workflow to recover from system faults using ULFM. We start with a fault affecting at least one MPI process. This fault is detected by any other living process with which the failed process has ongoing communication. This can happen for example during a collective operation like `MPI_Reduce`. The operation then raises an error of type `MPI_ERR_PROC_FAILED` (or `MPI_ERR_PROC_FAILED_PENDING` for non-blocking operations), which tells the application that subsequent function calls will not work unless treated separately. Afterwards, the processes aware of the failure *revoke* the communicator, indicating that it can no longer be used. As a next step, the failed processes can be excluded from the communicator by *shrinking* it. Optionally, the user can obtain certain information about the failed processes (like their ranks) via the `MPI_Comm_failure_ack` and `MPI_Failure_get_acked` functions. Finally, the surviving processes can synchronize the information about which processes have failed in the communicator.

The main advantage of ULFM is the ability to continue operation after faults occur by ensuring a consistent view of the affected communicators by all processes involved. This is a key functionality for ABFT, and one which we will exploit.

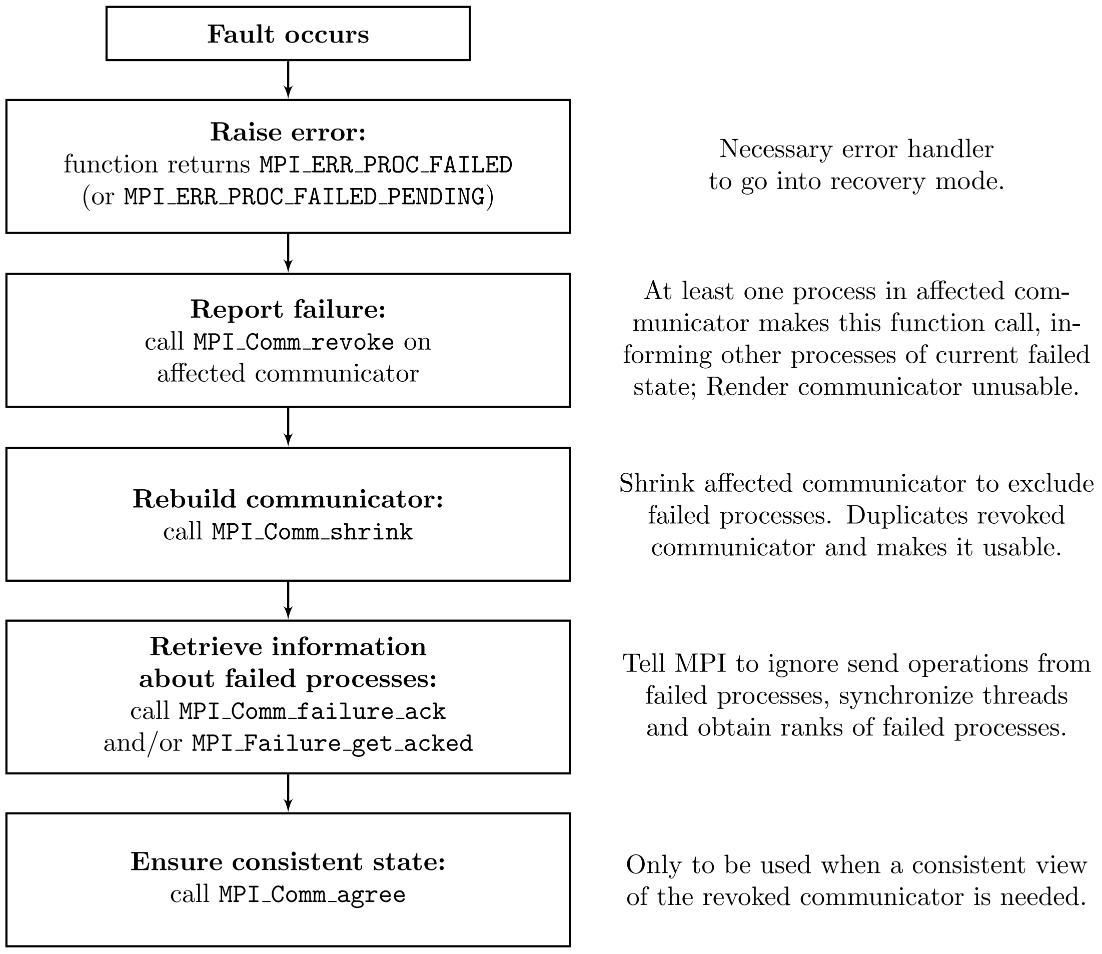

Fault occurs
Raise error: function returns MPI_ERR_PROC_FAILED (or MPI_ERR_PROC_FAILED_PENDING)
Necessary error handler to go into recovery mode.
Report failure: call MPI_Comm_revoke on affected communicator
At least one process in affected communicator makes this function call, informing other processes of current failed state; Render communicator unusable.
Rebuild communicator: call MPI_Comm_shrink
Shrink affected communicator to exclude failed processes. Duplicates revoked communicator and makes it usable.
Retrieve information about failed processes: call MPI_Comm_failure_ack and/or MPI_Failure_get_acked
Tell MPI to ignore send operations from failed processes, synchronize threads and obtain ranks of failed processes.
Ensure consistent state: call MPI_Comm_agree
Only to be used when a consistent view of the revoked communicator is needed.

Figure 5.5: Basic workflow of ULFM during faults (as described in [BBH$^+$13]).

The current ULFM specification, which is only roughly 20 pages long, is based on MPI 3.1 and is being evaluated by the MPI standardization body[3]. Until it is officially accepted into the MPI standard, ULFM cannot be found on native MPI installations of supercomputer centers. It can be installed locally but the user would have to choose between the native MPI implementation or the ULFM branch, and we believe the latter to be disadvantageous in terms of performance. For this reason we chose a compromise: we use the native MPI implementation of the HPC system at our disposal (which is not fault tolerant) and put an *additional* layer on top of it that emulates the functionalities of ULFM. This layer therefore implements the ULFM interface but it is not capable of dealing with real system faults. The main motivation is to have a code that can reproduce the behavior of ULFM with *simulated* faults, for example, by forcing certain ranks to go into infinite loops. This already allows us to perform many useful parallel tests with the only disadvantage of not being able to measure the overhead of the ULFM-specific functions (from Fig. 5.5). As we will see in our parallel experiments, we believe this overhead to be negligible with respect to other steps of the FTCT.

[3]http://fault-tolerance.org/ulfm/ulfm-specification/

Algorithm 4: Fault recovery in parallel

```
1 faultIDs = manager.getFaultIDs() ;                    // Get IDs of failed tasks
2 recomputeIDs, redistributeIDs = manager.solveGCP(faultIDs) ;
    // Solve GCP to find new combination technique
3 manager.recoverCommunicators() ;                      // Rebuild using MPI_Comm_shrink
4 manager.updateCombiParameters() ;                     // Inform groups of new coefficients
5 manager.recompute(recomputeIDs) ;                     // If necessary, recompute some tasks
6 manager.redistribute(redistributeIDs) ;               // Redistribute failed tasks to
    living groups
7 manager.combine() ;                                   // Combine solution with new coefficients
8 manager.restoreCombinationTechnique() ;               // Use original combination
    coefficients
```

Now we describe in more detail how we implemented the FTCT in the C++ parallel framework.

5.2.2 Parallel Implementation of the FTCT

We can now integrate the ideas behind ULFM with the theory of the FTCT to extend the parallel C++ framework described in Chapter 4 and make it tolerant to faults. We need two basic ingredients:

1. The parallel algorithms to compute new combination coefficients and combine according to the FTCT, and
2. The mechanisms to simulate, detect and recover from faults at the MPI level.

In the experiments of Section 5.1.1 we described how to perform the first step in serial. Now we want to see how this translates into a parallel implementation. The second step requires a ULFM-like interface to react to simulated faults.

Algorithms

Recall that our parallel framework from Chapter 4 is based on a manager-worker model using process groups as working agents. After the manager distributes the tasks to the groups, each group computes a certain number of time steps for all its tasks and waits for a signal to combine, first within the group and then across groups. We now assume that faults affect a subset of processes, which may be found in different process groups. Given that the vast amount of the simulation time is spent solving the PDE for a number of time steps (several orders of magnitude, as we will show in our experiments) we assume that hard faults occur at this stage with high probability.

Figure 5.6 illustrates this situation, and the steps to be followed in order to recover are summarized in Algorithm 4. These steps are to be carried out after the process groups have finished (or attempted to finish) computing their set of tasks, and a group returns the flag `PROCESS_GROUP_FAIL`, indicating that something went wrong. If a worker process fails, the master process will detect it during a call to `MPI_Barrier` within the group and it will inform the manager of this failure. If a master process fails, the manager will notice it after a call to `MPI_Wait` in the global communicator.

In Figure 5.6 we show a scenario where a fault has affected node 5 used by process group 2. (We could also assume that only one or several processes in the node have failed, or that several nodes have failed. The recovery procedure will look the same.) When this happens, the manager performs the following steps:

- Obtain the IDs of the tasks assigned to the failed groups (line 1)
- Solve the Generalized Coefficient Problem to find new combination coefficients (line 2). We used the library `GLPK` [glp] to solve the GCP using the optimization functions it offers, in particular the simplex algorithm to solve the minimization problem (3.32). Once the new coefficients are computed, the manager knows which tasks have to be recomputed(if any) by assigning them to living groups and which ones have to be only redistributed to living groups without recomputing.
- The manager then removes *the whole process group where faults occurred* from the global communicator (line 3). In theory one could remove only the affected processes in the group and use the remaining living processes for other purposes [4]. (Removing the whole group when, say, only one process fails, may sound like a waste of resources, but as we will see we can easily afford to do this and it doesn't represent a meaningful cost.)
- In the next step the manager informs the living process groups of the new combination coefficients (line 4).
- The manager then orders the living groups to recompute whichever tasks need to be recomputed, assigning them to any living groups (line 5).
- The rest of the failed tasks are redistributed to living groups without being recomputed (line 6). Any initialization routines needed for the tasks are also performed at this point.
- The combination step can take place with the alternative combination coefficients (line 7).
- For the next set of time steps the original combination technique can be used, so we restore the coefficients to the way they were originally defined (line 8).

After carrying out these steps, the combination technique can continue as originally, only with one process group less. This process can be repeated if more faults occur and it will keep working as long as there is at least one living process group.

[4]This is currently being tested in [OPHH$^+$].

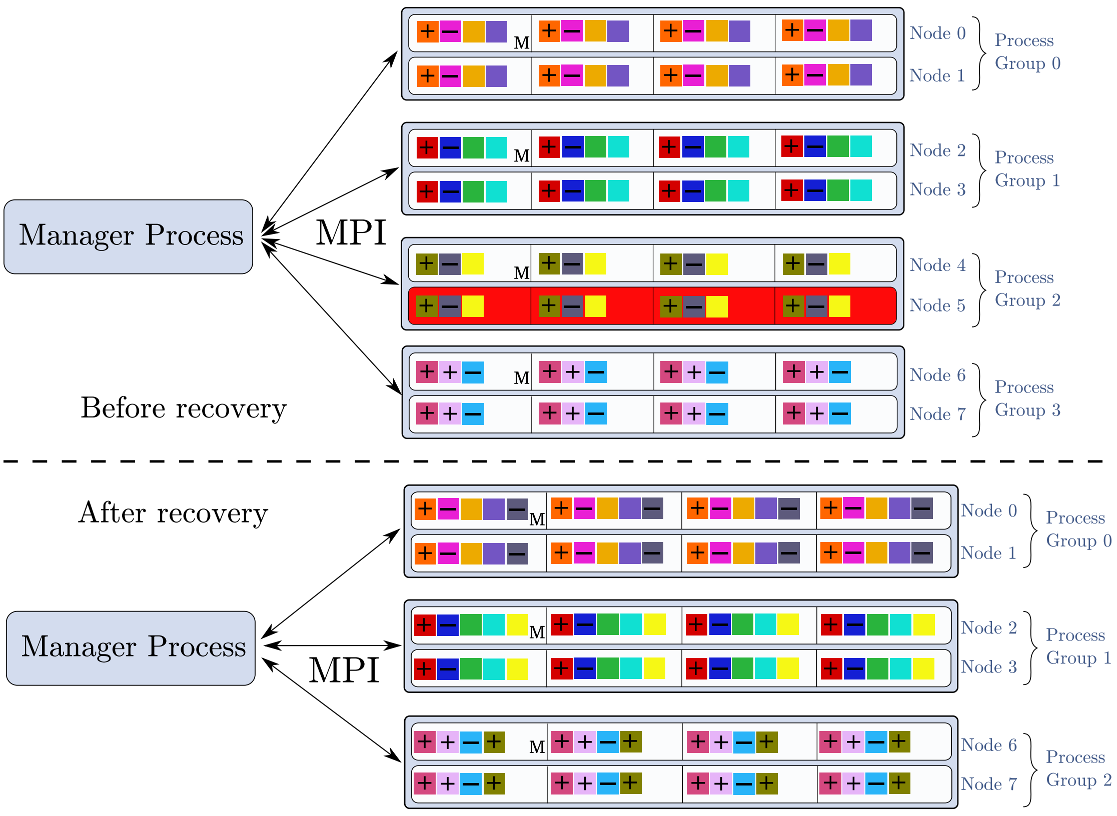


Figure 5.6: Communicators and process groups before and after recovering from a node failure (which here affects node 5).

One could interpret this implementation as a hybrid between algorithm-based fault tolerance and checkpointing. It is algorithm-based because it relies on the FTCT, which in theory does not require rolling backwards to a checkpoint. But the combination solution $u_n^{(c)}$ could be interpreted as a checkpoint for all component solutions $u_\mathbf{i}$, and in the cases where we need to recompute some $u_\mathbf{i}$, these are effectively rolling back to the last stored solution $u_n^{(c)}$.

ULFM Interface

So far we have left out the details of our MPI implementation and focused on the algorithms. As we mentioned earlier, ULFM is currently not available natively in most HPC systems. Based on the assumption that installing ULFM manually would lead to performance losses, we decided to settle for a compromise and implemented an *interface* that behaves exactly like ULFM but calls regular MPI functions in the background. This means that the code can't recover from real faults but it can replicate the behavior of ULFM. Once ULFM becomes available in HPC systems, our interface can be switched with the real implementation. A detailed description of the interface can be found in [Wal16].

A process failure is simulated by calling a special `Kill_me` function at any point of the program, and the corresponding process stops participating in the application. For our initial tests we specify the number of faults that will occur during the simulation, the iterations at which the faults will occur, and the MPI ranks that will be affected. We do this by extending the settings file by an additional `faults` section that looks as follows:

```
[faults]
num_faults = 2
iteration_faults = 10 20
global_rank_faults = 3 8
```

This would simulate two faults at iterations 10 and 20, affecting ranks 3 and 8 respectively.

The most important blocking and non-blocking MPI functions are redefined in the interface in a way that they can react to dead processes. Additionally, the interface defines all ULFM functions listed in diagram 5.5. Although the layer is not optimized for performance, it has a low overhead compared to a native MPI implementation. But most importantly, we will see that the main overhead of the FTCT comes from functions other than MPI functions (by a considerable factor).

5.2.3 Simulation Scenario

To test our parallel implementation of the FTCT we opted for a d-dimensional advection-diffusion equation implemented in DUNE-pdelab. The PDE is given by

$$\begin{aligned} \partial_t u - \Delta u + \mathbf{a} \cdot \nabla u &= f \qquad \text{in } \Omega \times [0, T) \\ u(\cdot, t) &= 0 \qquad \text{in } \partial\Omega \end{aligned} \tag{5.3}$$

with $\Omega = [0,1]^d$, $\mathbf{a} = (1, 1, ..., 1)^T$ and $u(\cdot, 0) = e^{-100 \sum_{i=1}^{d} (x_i - 0.5)^2}$. The space is discretized with the finite volume element method on rectangular d-dimensional grids. This method uses node-centered control volumes, as opposed to the classical finite volume method where the degrees of freedom are positioned at the cell corners. Our integration scheme in time is a simple explicit Euler method.

We carry out two sets of experiments, analogously to our serial tests with `GENE` . First we investigate the approximation quality of the combination solution after recovering from faults. Given the positive results with `GENE` , we can expect the solution of this simpler model to be accurate as well. As with `GENE` , we use a reference solution of high level to compare the combination technique solution with and without faults. In all our experiments we vary the number of process groups and we randomly choose one of them to fail. It makes little difference which process group fails since the tasks are evenly distributed among the groups. We investigate the quality of the solution for both $d = 2$ and $d = 5$. In the 2D case we use a truncation level of $\boldsymbol{\tau} = (2, 2)$ and increase the maximum level from $\mathbf{n} = (6, 6)$ up to $\mathbf{n} = (10, 10)$. For each level of resolution we compare the combination technique with a full grid solution of level $\mathbf{n} = (11, 11)$. We use a time step of $\Delta t = 10^{-4}$ and run 1,000 time steps (until $t = 0.10$), combining after every time step. In five dimensions we use $\Delta t = 10^{-3}$ and perform 100 time steps,

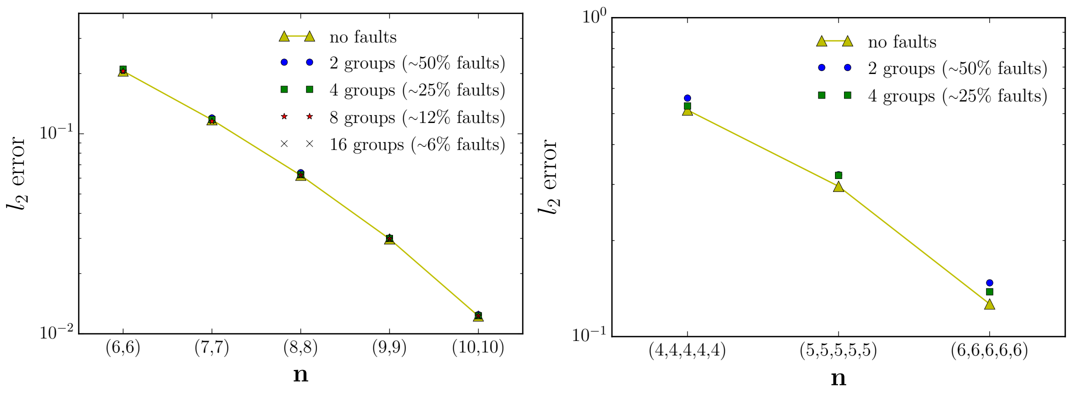


Figure 5.7: Convergence results for the advection-diffusion equation implemented in DUNE in 2D (left) and 5D (right)[5].

using a truncation parameter of $\boldsymbol{\tau} = (2, 2, 2, 2, 2)$ and increasing the resolution from $\mathbf{n} = (4, 4, 4, 4, 4)$ to $\mathbf{n} = (6, 6, 6, 6, 6)$. The full grid reference solution has level $\mathbf{n} = (6, 6, 6, 6, 6)$ as well. In both 2D and 5D cases we calculate the relative l_2-error (5.1) at the end of the simulation.

In the second set of tests we are interested in whether the FTCT scales well, particularly the steps that are specific to the FTCT: redistributing the lost tasks and recomputing some tasks whenever necessary. We do this for a five-dimensional example with $\mathbf{n} = (8, 8, 7, 7, 7)$ and $\boldsymbol{\tau} = (4, 4, 3, 3, 3)$, which results in 126 component grids. It is worth noting that it would not be possible to compute the full grid solution of level $\mathbf{n} = (8, 8, 7, 7, 7)$ even on the full supercomputer. We use 8192, 16384, 32768 and 65536 processes distributed among 8, 16 and 32 process groups, each with 512, 1024, 2048 or 4096 processes.

Process failures are simulated by calling the `Kill_me` function introduced earlier, which causes the calling process to stop reacting in any MPI calls. All tests were performed on the supercomputer *Hazel Hen* at HLRS Stuttgart. This system counts with 7,712 Intel Xeon CPU E5-2680 v3 compute nodes, each of which has two sockets with 12 cores each (185,088 cores in total) and 128 GB of memory. To the best of our knowledge, these are the first massively parallel experiments in more than three dimensions with the combination technique.

5.2.4 Results

Convergence

Figure 5.7 shows our convergence results in two (left) and five (right) dimensions. In all cases we insert a process fault at the second time iteration, since we noticed virtually no difference on the quality of the solution as a function of the iteration where the fault occurs. We also show the results for single runs instead of averaging over many runs, for two reasons: 1) many of the simulation scenarios are quite expensive to calculate (especially in 5D), and 2) since the tasks are distributed

[5]Figure as published in [HPHBP16].

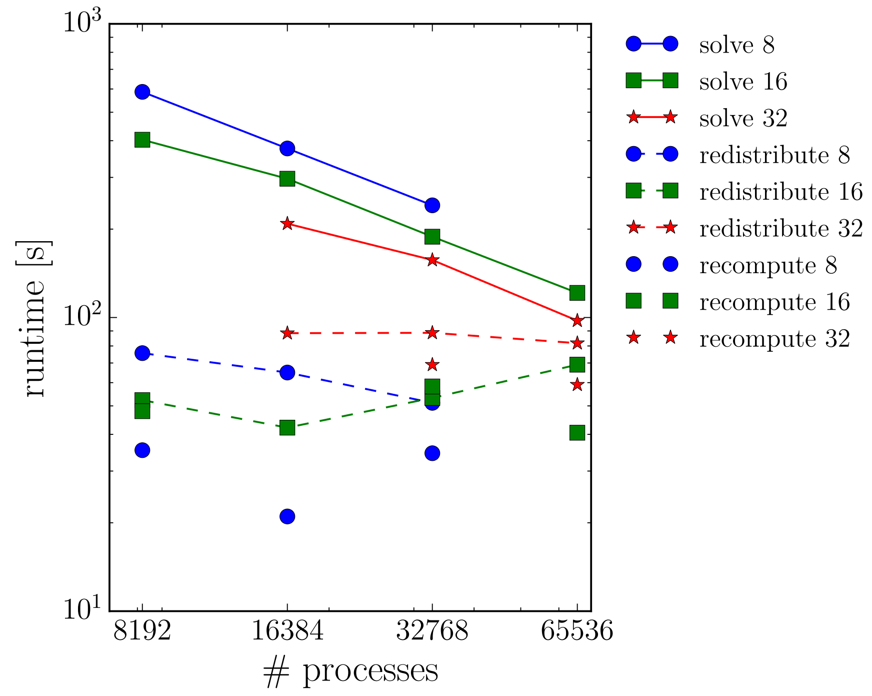


Figure 5.8: 5D scaling experiments in DUNE. The numbers 8, 16 and 32 indicate the number of process groups used[6].

uniformly across groups, the tasks that end up being assigned to a given group are practically random, so this already introduces some variance.

As we can see from the plots, the combination technique with faults is only slightly worse than when no faults occur. The difference is in the order of 1%–3%, even when half the tasks fail. Remember that in each case we remove all tasks belonging to the group with the failed process. These results confirm our observations with `GENE`: the FTCT follows the classical combination technique closely in terms of the error, even for a considerable percentage of failed tasks.

These result raise an obvious question: if the combination technique after faults occur is so close to that without faults, why not use a combination of lower resolution from the beginning? It is important to keep in mind that sparse grids are obtained by finding the optimal compromise between lowering the total number of degrees of freedom and keeping the error low. Moving away from this optimum by decreasing (or increasing) the number of degrees of freedom results in a higher cost in terms of the approximation error than what is gained by the fewer degrees of freedom. Additionally, the alternative combination coefficients used after faults usually still include some of the high-resolution grids (those for which $|\mathbf{l}|_1$ is highest), but this would not be the case anymore if we decreased the reference level $\mathbf{n}$ uniformly to $\mathbf{n}' = \mathbf{n} - \mathbf{1} \cdot c$.

[6]Figure as published in [HPHBP16].

Scalability

Figure 5.8 shows three different steps of the algorithm that were timed, namely

- *solve*, the time it takes the DUNE solver to perform one time step of the PDE,
- *redistribute*, the time needed to redistribute failed tasks to living groups, including any initialization routines, and
- *recompute*, the time it takes the algorithm to recompute any failed tasks, if necessary.

We also measured the time to combine the solutions and the to calculate the alternative combination coefficients, but in all cases these contributions were negligible.

As with the convergence experiments, we only show the results for one run, instead of the average over several runs. Once again this is due to the high cost of running each simulation (requiring several hours in the most expensive experiments). In this case, the results look more erratic, but the orders of magnitude are still informative. What we see is that redistributing and recomputing are up to one order of magnitude less expensive than performing one single time step of DUNE. This is a good result, especially considering that we expect the user to combine only every few time steps and not after every time step. It is also worth mentioning that most of the time measured under *recomputing* is due to the initialization routines that have to be called for every task, which happen to be quite slow in DUNE. Also, the initialization doesn't seem to scale, which explains the increase in the green dashed curve.

We are enthusiastic about these results, since they indicate that the convergence of the FTCT is good and the overhead is very much affordable.

5.3 Outlook: Parallel Simulations with GENE

The next natural step in our research is to test our parallel framework with a real-life application, such as `GENE`. As of this writing, `GENE` has been successfully coupled to the parallel framework for the combination technique [Hee17]. This is not a straightforward task, since it requires adapting the boundary conditions as required by the combination technique. Additionally, it is not clear how often one should combine the component solutions. Whether one should combine very often (say, at every time step) or rarely (for example, only once at the end of the simulation) seems to depend strongly on the physical scenario and the choice of n and $\boldsymbol{\tau}$. But once a suitable combination interval is chosen (along with a high enough $\boldsymbol{\tau}$), the combination technique seems to converge quite well.

With a working parallel version of the combination technique coupled to `GENE` , we were able to apply the FTCT and run similar tests to those of Section 5.2

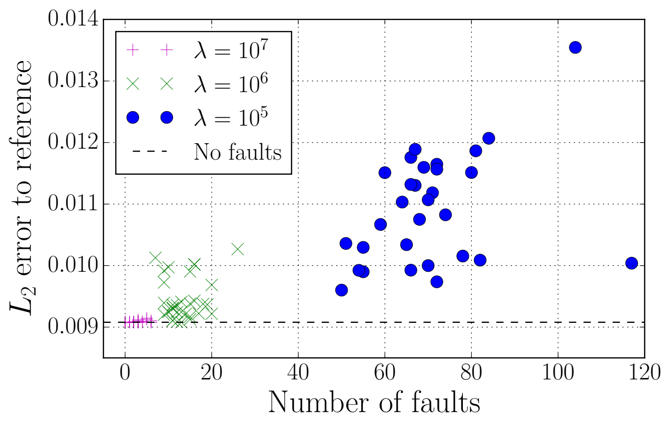


Figure 5.9: Error of the combination technique with `GENE` as a function of the total number of faults affecting the simulation. Different fault frequencies λ are used for the Weibull distribution[8].

[OPHH$^+$]. Two main new features were introduced[7]: 1) faults can be injected according to any probability distribution function, and 2) if only a few processes in a group are affected by faults, they can be replaced by spare processors (if there are any available) instead of getting rid of the whole group. The first features allows us to emulate different type of fault scenarios and obtain more robust error statistics. And by replacing dead processors with spare ones, we can tolerate more faults, since it is less likely that all process groups are removed due to failures, which would cause the simulation to crash.

Figure 5.9 shows some parallel results using a Weibull distribution function

$$f(t, \lambda, k) = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-(t/\lambda)^k}, \tag{5.4}$$

with $k = 0.7$ and different values of the mean time between failures λ. The reference error for this scenario (a linear simulation as in Section 5.1.1) is 0.009, so we can see that the combination solution with faults is quite close to the reference. It takes a really large number of faults to start noticing a larger deviation from the reference, but even for simulations where a very large number of hard faults hit the system, the error is quite close to the reference solution. All simulations were performed using four process groups and 512 cores in total.

Figure 5.10 shows some preliminary scaling results for the most expensive steps of the FTCT. To generate these results, we used a combination technique with $n = 10$ and $\boldsymbol{\tau} = (3, 3, 3)$ in dimensions $(z, \mu, v_{\parallel})$, keeping the number of discretization points in x and y constant (to 513 and 1, respectively). The classical combination technique (not fault tolerant) would have in this case 136 component grids. The fault tolerant combination technique requires 49 component grids more (for a total of 185). This causes the time to solve to increase (blue line vs. green

[7]Implemented by Michael Obersteiner

[8]Figure as submitted in [OPHH$^+$] with data by Michael Obersteiner.

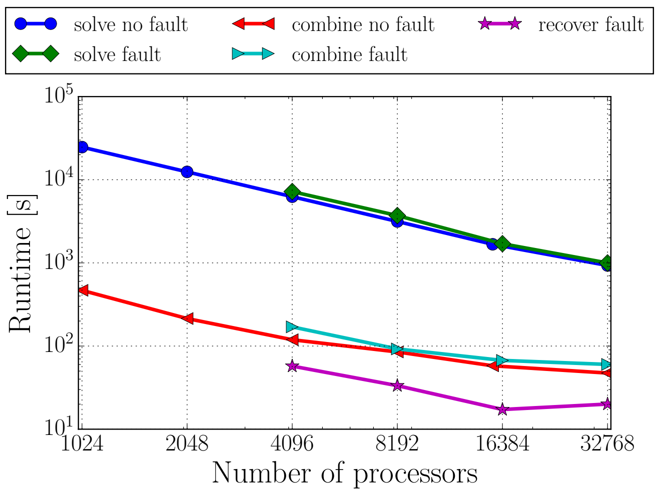


Figure 5.10: Scaling of the different steps of the parallel fault tolerant combination technique with `GENE`. The time to recover from faults is several orders of magnitude smaller than the time it takes to compute a single time step of the application[9].

line), but only slightly. The time to recover from faults is roughly two orders of magnitude smaller than the time to solve one time step, and it seems to scale well at least up to 16k cores. Afterwards there is a small increase, but this was observed to be rather due to high variations in runtime from simulation to simulation – in some cases, the time was much smaller. At this scale, however, it is difficult to perform many runs in order to get more robust statistics.

The final step in our research would involve simulating more complex physical scenarios (in global mode with nonlinear effects) in order to gain new insights into the turbulence phenomena. It is still unclear how well the combination technique will be able to deal with nonlinear simulations, which might also have consequences for the fault tolerance algorithms we have studies so far. However, our experiments have consistently confirmed that whenever the combination technique works well, the FTCT also converges.

[9]Figure as submitted in [OPHH$^+$] with data by Michael Obersteiner.

6 Dealing with Silent Errors with the Combination Technique

In this chapter we want to address the problem of silent errors (and, in particular, *silent data corruption*, or SDC) described in Chapter 2.3. In particular, we want to answer the following questions:

- How can silent errors affect the combination technique?
- How can we model silent errors – and in particular, SDC – realistically?
- Is it possible to detect and correct SDC that has affected the combination technique?
- Can we recover from SDC in a way that is cheap and possibly scalable?

Our aim for this chapter is to show that the fault tolerant combination technique can also be used to recover from SDC, with some additional work. The approach we will follow is purely algorithmic, and we will try to show that this is a very promising ansatz for silent error resilience.

6.1 SDC and the Combination Technique

How can silent errors affect the combination technique? The answer is: we don't really know. As we mentioned in Chapter 2, silent errors, and in particular SDC, are still poorly understood. For this reason, we will not try to make any overly specific assumptions regarding the nature of SDC, and will use a very general model of SDC with which we hope to cover as many fault scenarios as possible. This means, for instance, not assuming that SDC will occur exclusively as bit flips, since there could be other ways in which the floating-point data could be corrupted. Also, injecting random bit flips into an application code (a common approach in the community) might not be representative of the application's behavior, since random bit flips would tend to affect the mantissa and not the exponent or the sign. In the end, what we should be concerned with are numerical errors in the data, independently of what causes them. If we formulate the problem in terms of numerical errors, we can use classical approaches in numerical mathematics and talk in terms of norms and bounds.

Let us now discuss ways to model SDC in a way that we avoid making unnecessary assumptions. We follow the methodology described in Chapter 2, as proposed in [EHM14a], which consists in selectively manipulating the data in the most critical steps of the algorithm and altering the data in a way that covers all possible orders of magnitude. As opposed to injecting random bit flips, this approach can tell us how the application might behave in the worst case scenarios, which is what we should aim for. Implementing this involves identifying the parts of the algorithm where we cannot afford SDC to propagate, meaning we should by all means be able to detect and correct any anomalous data at those critical steps. There could be other parts of the algorithm where it might not be necessary to detect SDC, so no computational resources should be spent there. This approach is called *selective reliability* [BFHH12].

With this in mind we can once again analyze the various steps combination technique, given in algorithm 5. As we have argued, most of the computation time is spend solving the PDE (line 6). Likewise, the vast majority of the memory requirements are used to store the component solutions $u_{\mathbf{i}}$. The only strong assumption we will make is that, if SDC occurs, it will be reflected in the data corresponding to the component solutions. Especially as we increase the dimension of the PDE, each solution might take up several gigabytes of data. We think it is reasonable to focus on SDC affecting floating-point data as opposed to including *metadata* as well – pointers, indices, counters or instructions – for the reasons we discussed in Chapter 2.

If we focus on errors on the floating-point data, we can ask which parts of the algorithm should be performed reliably. Keep in mind that each component solution is computed, hierarchized and dehierarchized independently of each other. This means that if SDC affects a component solution at any of these steps, it will not propagate to the other solutions, so we can arguably perform these steps without checking for correctness. The only crucial step is the combination step (line 6), since the results from all component solutions are added to the sparse grid solution, which is then used as initial condition for the next set of time steps. We argue that this step should be performed reliably, that is, any SDC should be identified and excluded before combining. Allowing SDC to propagate to other component solutions could ruin the whole combination, and we will see in our tests that this is indeed the case. Additionally, the combination step is a linear operation, so the error introduced by a wrong component solution will have the same order of magnitude in the sparse grid solution. We will tolerate errors in all other stages.

In line 8 of the algorithm we can add a function to check for any errors in the component solutions. This function should ideally tell us which component solutions (if any) might have been affected by SDC. $\mathcal{J}_{\text{sdc}}$ denotes the set of indices corresponding to the affected solutions. If we manage to identify them, we can exclude them from the combination, following the same steps as when dealing with hard faults. The function `SanityCheck` should 1) be inexpensive, 2) reliably find SDC and 3) be based on error bounds specific to the combination technique, not to the actual simulation code being used as solver. This chapter aims to describe

Algorithm 5: Truncated combination technique with sanity check for SDC

```
input : A function solver; sparse grid resolution $n$; parameter $\boldsymbol{\tau}$; time
        steps per combination $N_t$
output: Combined solution $u^{(c)}_{n,\boldsymbol{\tau}}$
1  Generate index set $\mathcal{I}^{d,n}_{\boldsymbol{\tau}} = \bigcup_{q=0}^{d-1} \mathcal{I}^{d,n}_{q,\boldsymbol{\tau}}$ and compute coefficients $c_{\mathbf{i}}$;
2  for $\boldsymbol{i} \in \mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ do
3      $u_{\mathbf{i}} \leftarrow u(\vec{x}, t = 0)$ ;                         // Set initial conditions
4  while not converged do
5      for $\boldsymbol{i} \in \mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ do
6          $u_{\mathbf{i}} \leftarrow$ solver($u_{\mathbf{i}}$,$N_t$) ;        // Solve the PDE on grid $\Omega_{\mathbf{i}}$ ($N_t$ time steps)
7          $u_{\mathbf{i}} \leftarrow$ hierarchize($u_{\mathbf{i}}$) ;          // Transform to hier. basis, Eq. (3.9)
8      $\mathcal{J}_{\text{sdc}} \leftarrow$ SanityCheck({$u_{\mathbf{i}}$ }) ;           // Check for SDC in all $u_{\mathbf{i}}$
9      if $\mathcal{J}_{sdc}$ not empty then                               // Did SDC affect any $u_{\mathbf{i}}$?
10         {$c_{\mathbf{i}}$} $\leftarrow$ computeNewCoeffs($\mathcal{J}_{sdc}$) ;      // Update combination coeffs.
11     $u^{(c)}_{n,\boldsymbol{\tau}} \leftarrow$ combine($c_{\boldsymbol{i}} u_{\mathbf{i}}$) ;        // Combined solution (in the hier. basis)
12     $u^{(c)}_{n,\boldsymbol{\tau}} \leftarrow$ dehierarchize($u^{(c)}_{n,\boldsymbol{\tau}}$) ;        // Transform back to nodal basis
13     for $\boldsymbol{i} \in \mathcal{I}^{d,n}_{\boldsymbol{\tau}}$ do
14         $u_{\mathbf{i}} \leftarrow$ sample($u^{(c)}_{n,\boldsymbol{\tau}}$) ;          // Sample each $u_{\mathbf{i}}$ from new $u^{(c)}_{n,\boldsymbol{\tau}}$
```

some possible sanity checks and test them both in serial and in parallel.

6.2 Preliminary Studies

Let us start once again with the linear 2D advection equation introduced in Chapter 3.2,

$$\frac{\partial u}{\partial t} + c_x \frac{\partial u}{\partial x} + c_y \frac{\partial u}{\partial y} = 0 \tag{6.1}$$

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. We could then try solve this PDE using Algorithm 5, but now we allow SDC to affect one or more component solutions at any step of the algorithm. In order to determine if any of the solutions is wrong, we need to define what magnitude of error we are willing to tolerate and how we are to determine if an error falls outside that threshold. To see this more clearly, imagine we randomly choose one of the component solutions and alter one of its values by one of the following factors:

1. $\tilde{u}_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}) = u_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}) \times 10^{-300}$ (very small)
2. $\tilde{u}_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}) = u_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}) \times 10^{-0.5}$ (slightly smaller)
3. $\tilde{u}_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}) = u_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}}) \times 10^{+150}$ (very large)

This could happen, for example, during the call to the `solve` function, but not exclusively. Any other step apart from `combine` (for example, hierarchization) could also be subject to faults. How can we determine whether a component solution has been affected? Evidently, we don't what the exact solution will look like, but we do have a set of solutions of the same PDE with different discretizations, so whatever the actual solution looks like, we should expect all component solutions to look somewhat similar. If one of them happens to be very different from the rest, we could use this as an indication that it might be wrong. To achieve this, we propose using the theory of the combination technique and translate this heuristic into error bounds. We now describe two possible implementations of the `SanityCheck` function.

6.2.1 Sanity Check 1: Comparing Combination Solutions Pairwise via the Maximum Norm

Recall the error splitting assumption (ESA) in two dimensions introduced in Chapter 3.2, which states that every component solution should satisfy the relation

$$u - u_{\mathbf{i}} = C_1(\vec{x}, h_{i_1})h_{i_1}^p + C_2(\vec{x}, h_{i_2})h_{i_2}^p + C_{1,2}(\vec{x}, h_{i_1}, h_{i_2})h_{i_1}^p h_{i_2}^p. \tag{6.2}$$

where $p \in \mathbb{N}$ and the functions C_1, C_2 and $C_{1,2}$ are bounded, $|C_1| \leq \kappa_1(\vec{x})$, $|C_2| \leq \kappa_2(\vec{x})$ and $|C_{1,2}| \leq \kappa_{1,2}(\vec{x})$.

Let us now see what happens in the hierarchical basis. From the definition of the hierarchical surpluses (3.10) it is clear that they also satisfy the ESA:

$$\alpha_{\mathbf{l},\mathbf{j}} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{i})} = D_1(x_{\mathbf{l},\mathbf{j}}, h_{i_1})h_{i_1}^p + D_2(x_{\mathbf{l},\mathbf{j}}, h_{i_2})h_{i_2}^p + D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{i_1}, h_{i_2})h_{i_1}^p h_{i_2}^p, \tag{6.3}$$

where $\alpha_{l,j}$ is the exact surplus at grid point $x_{l,j}$. The functions appearing at each term are also bounded, $|D_1| \leq 4\kappa_1(x_{\mathbf{l},\mathbf{j}})$, $|D_2| \leq 4\kappa_2(x_{\mathbf{l},\mathbf{j}})$, and $|D_{1,2}| \leq 4\kappa_{1,2}(x_{\mathbf{l},\mathbf{j}})$. This follows from the definition of the two-dimensional hierarchization operator:

$$\begin{aligned}
\alpha_{\mathbf{l},\mathbf{j}} &= \left(\prod_{k=1}^{2} \begin{bmatrix}-\tfrac{1}{2} & 1 & -\tfrac{1}{2}\end{bmatrix}_{l_k,j_k}\right) u(x_{l_1,j_1}, x_{l_2,j_2}) \\
&= \left(\begin{bmatrix}-\tfrac{1}{2} & 1 & -\tfrac{1}{2}\end{bmatrix}_{l_2,j_2}\right) \left(u(x_{l_1,j_1}, x_{l_2,j_2}) - \frac{u(x_{l_1,j_1-1}, x_{l_2,j_2}) + u(x_{l_1,j_1+1}, x_{l_2,j_2})}{2}\right) \\
&= u(x_{l_1,j_1}, x_{l_2,j_2}) \\
&\quad - \frac{u(x_{l_1,j_1}, x_{l_2,j_2-1}) + u(x_{l_1,j_1}, x_{l_2,j_2+1}) + u(x_{l_1,j_1-1}, x_{l_2,j_2}) + u(x_{l_1,j_1+1}, x_{l_2,j_2})}{2} \\
&\quad + \frac{u(x_{l_1,j_1-1}, x_{l_2,j_2-1}) + u(x_{l_1,j_1-1}, x_{l_2,j_2+1}) + u(x_{l_1,j_1+1}, x_{l_2,j_2-1}) + u(x_{l_1,j_1+1}, x_{l_2,j_2+1})}{4}.
\end{aligned}$$

For $\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{i})}$, we would have the same expression but with $u_{\mathbf{i}}$ instead of u. Taking the difference of both and using (3.21), we obtain (6.3).

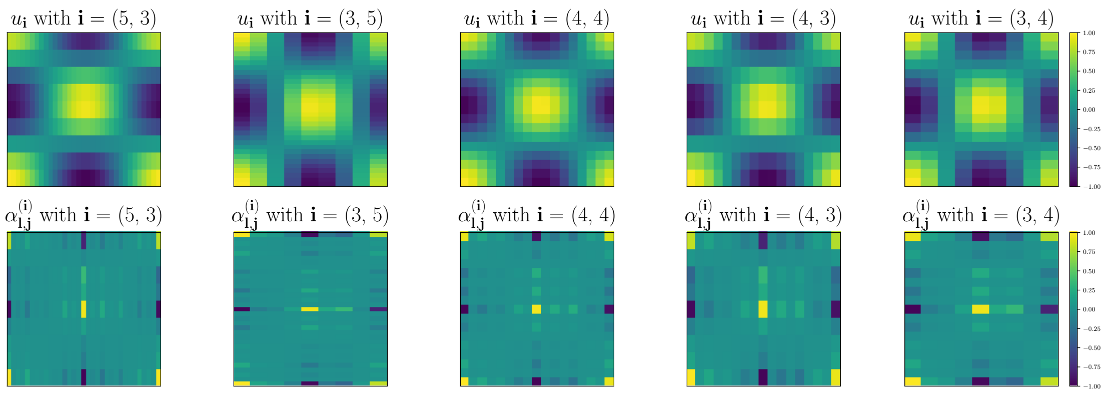


Figure 6.1: Five component solutions of the 2D advection equation with their respective representation in the hierarchical basis.

As a next step we take two arbitrary component solutions in two dimension, say $u_{\mathbf{s}}$ and $u_{\mathbf{t}}$, $\mathbf{s}, \mathbf{t} \in \mathcal{I}$. Since each satisfies Eq. (6.3), their difference satisfies

$$\begin{aligned}\alpha^{(\mathbf{t})}_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{s})}_{\mathbf{l},\mathbf{j}} &= D_1(x_{\mathbf{l},\mathbf{j}}, h_{t_1})h^p_{t_1} + D_2(x_{\mathbf{l},\mathbf{j}}, h_{t_2})h^p_{t_2} + D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{t_1}, h_{t_2})h^p_{t_1}h^p_{t_2} \\ &\quad - D_1(x_{\mathbf{l},\mathbf{j}}, h_{s_1})h^p_{s_1} - D_2(x_{\mathbf{l},\mathbf{j}}, h_{s_2})h^p_{s_2} - D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{s_1}, h_{s_2})h^p_{s_1}h^p_{s_2}.\end{aligned} \tag{6.4}$$

Since solutions $u_{\mathbf{s}}$ and $u_{\mathbf{t}}$ are defined on grids with different resolutions, their difference can only be computed on the common grid $\Omega_{\mathbf{s}\wedge\mathbf{t}}$ composed of the subspaces $W_{\mathbf{l}}$ with $(1, 1) \le \mathbf{l} \le \mathbf{s} \wedge \mathbf{t}$. What Eq. (6.4) tells us is that the difference between the surpluses of two component solutions depends mostly on the individual grid resolutions h_i (dominated by the four univariate terms) and how similar their resolutions are, meaning that the smaller the distance $|\mathbf{t} - \mathbf{s}|_1$, the smaller the difference will be. This can be seen more easily if we rewrite (6.4) as

$$\begin{aligned}\alpha^{(\mathbf{t})}_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{s})}_{\mathbf{l},\mathbf{j}} =&(h^p_{t_1} - h^p_{s_1})D_1(x_{\mathbf{l},\mathbf{j}}, h_{s_1} - h_{t_1}) + (h^p_{t_2} - h^p_{s_2})D_2(x_{\mathbf{l},\mathbf{j}}, h_{s_2} - h_{t_2}) + \\ &\mathcal{O}(h^{p+1}_{s_1} + h^{p+1}_{t_1} + h^{p+1}_{s_1} + h^{p+1}_{s_2}),\end{aligned} \tag{6.5}$$

so the more similar $h_{\mathbf{s}}$ and $h_{\mathbf{t}}$ are, the smaller the difference in their hierarchical coefficients. (Eq. (6.4) can be recovered by taking the Taylor expansion of each function $D_i(h_{s_i} - h_{t_i})$.) It is straightforward to show that Eq. (6.4) is point-wise bounded by

$$\beta^{(\mathbf{s},\mathbf{t})}_{\mathbf{l},\mathbf{j}} := |\alpha^{(\mathbf{t})}_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{s})}_{\mathbf{l},\mathbf{j}}| \le 4 \cdot \kappa(x_{\mathbf{l},\mathbf{j}}) \cdot (h^p_{t_1} + h^p_{s_1} + h^p_{t_2} + h^p_{s_2} + h^p_{t_1}h^p_{t_2} + h^p_{s_1}h^p_{s_2}), \quad \mathbf{l} \le \mathbf{s} \wedge \mathbf{t}. \tag{6.6}$$

This could provide a starting point to detect SDC. We could imagine measuring the quantity $\beta^{(\mathbf{s},\mathbf{t})}_{\mathbf{l},\mathbf{j}}$ for many pairs of component solutions, and if for some pairs the difference exceeds the bound, this could be an indication that one of the component solutions is wrong. Unfortunately, we cannot apply bound (6.6) directly: it depends on the function κ, which is not known. It could be approximated if all component solutions have been computed correctly (to see how to do this, cf. [Har16a]), which is not the case if SDC affects one or more $u_{\mathbf{i}}$.

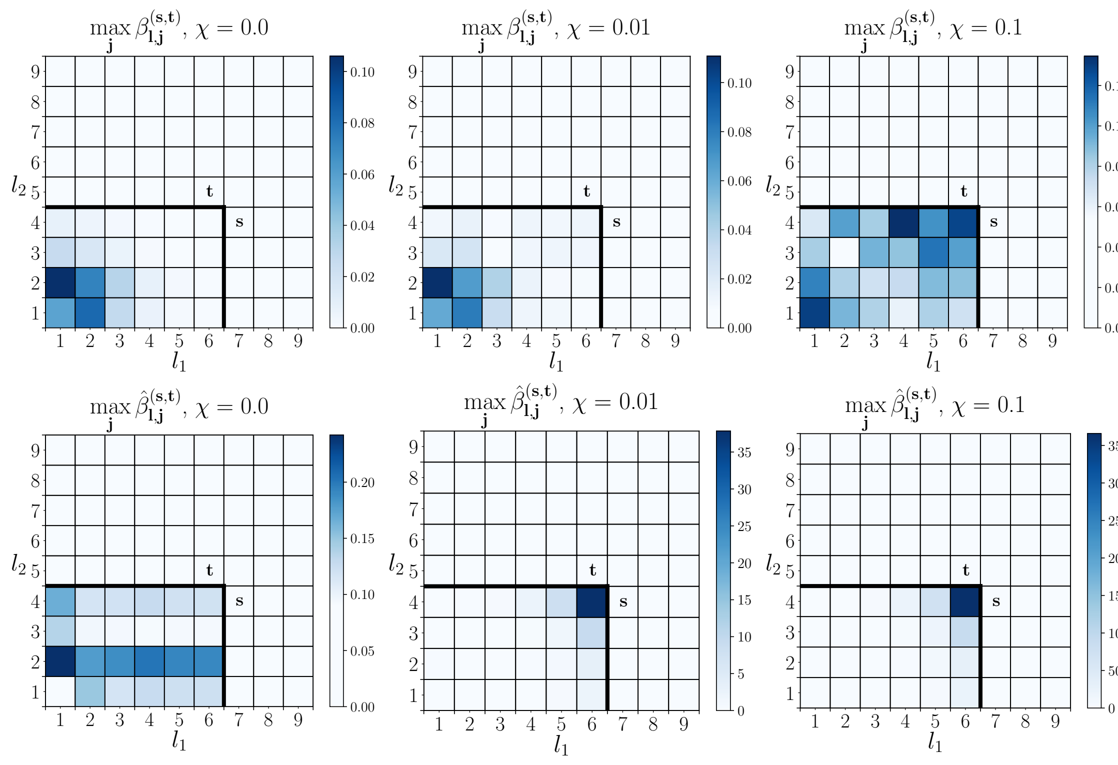


Figure 6.2: Upper row: Maximum difference of the hierarchical surpluses of two grids for each common subspace when noise of different magnitudes is present. Lower row: same scenarios, but normalizing the difference.

But the quantity $\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ might still be useful to detect SDC. Suppose we introduce some random noise in one of the combination grids, say $u_{\mathbf{s}}$,

$$\tilde{u}_{\mathbf{s}} = u_{\mathbf{s}} + \gamma \epsilon_{\mathbf{s}}, \tag{6.7}$$

where γ is a positive constant and $\epsilon_{\mathbf{s}}$ is a matrix of the same size as $u_{\mathbf{s}}$ with random entries uniformly distributed between 0 and 1. We can choose γ to vary how large the noise is, for example, by considering

$$\chi := \frac{\|\tilde{u}_{\mathbf{s}} - u_{\mathbf{s}}\|_2}{\|u_{\mathbf{s}}\|_2}. \tag{6.8}$$

χ tells us how much the noisy solution $\tilde{u}_{\mathbf{s}}$ differs from the computed solution $u_{\mathbf{s}}$. For example, $\chi = .05$ means $\tilde{u}_{\mathbf{s}}$ differs by 5% relative to $u_{\mathbf{s}}$. γ is then computed as $\gamma \equiv 2\chi \|u_{\mathbf{s}}\|_2 / \sqrt{\prod_k (2^{h_k} + 1)}$. We can observe what happens to $\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ when we compare solutions $u_{\mathbf{s}}$ and $u_{\mathbf{t}}$ with $\mathbf{s} = (7, 4)$, $\mathbf{t} = (6, 5)$, and noise is added to $u_{\mathbf{s}}$, with $\chi = 0.001, 0.01$ (.1% and 1% resp.). These were calculated at time $t = 0.25$ with velocity $\mathbf{a} = (0.5, 0.5)$. The top row of figure 6.2 shows the largest value of $\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ on each hierarchical subspace $W_{\mathbf{l}}$, $(1, 1) \leq \mathbf{l} \leq (6, 4)$. The values start to look increasingly random as χ increases, but the largest orders of magnitude are kept the same (as can be seen from the values of the color bar). But if we normalize

$\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ as follows

$$\hat{\beta}_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})} := \frac{|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|}{\min\left\{|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}|, |\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|\right\}} \quad \text{for all } \mathbf{l} \le \mathbf{s} \wedge \mathbf{t}, \quad \mathbf{0} \le \mathbf{j} \le 2^{\mathbf{l}}, \tag{6.9}$$

we can easily identify on the highest subspaces that something has gone wrong. This can be seen in the lower row of figure 6.2, where the value of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ increases by 1-2 orders of magnitude on the highest subspaces. This is due to the fact that, for interpolation problems, the surpluses decay exponentially [BG04]

$$|\alpha_{\mathbf{l},\mathbf{j}}| \le 2^{-d} \cdot \left(\frac{2}{3}\right)^{d/2} \cdot 2^{-(3/2)\cdot|\mathbf{l}|_1} \cdot \left\|D^{\mathbf{2}}(u|_{\text{supp}\ \phi_{\mathbf{l},\mathbf{j}}})\right\|_{L_2}, \tag{6.10}$$

with $D^{\mathbf{2}}(u) := \frac{\partial^4 u}{\partial x_1^2 \partial x_2^2}$. This is the motivation behind working in the hierarchical basis.

The reason we choose to divide by the smallest coefficient is that, if no SDC occurs, $|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|$ and $|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}|$ should have roughly the same value, but if SDC does occur, then one of the two values will be much larger than the other, so dividing by the smaller one will amplify the value of $\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$. As a final step we take the largest $\hat{\beta}_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}$ over all grid points $x_{\mathbf{l},\mathbf{j}}$,

$$\hat{\beta}^{(\mathbf{s},\mathbf{t})} := \max_{\mathbf{l}\le\mathbf{s}\wedge\mathbf{t}} \ \max_{\mathbf{j}\in\mathcal{I}_{\mathbf{l}}} \hat{\beta}_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})}. \tag{6.11}$$

In Algorithm 6 we summarize this implementation of the function `SanityCheck`. To show how the algorithm could work, we solved the 2D advection equation (6.1) using a combination technique with $\mathbf{n} = (9, 9)$ and $= (\mathbf{6}, \mathbf{6})$. This gives six combination grids. We simulated 129 time steps, until $t = 0.25$, with $c_x = c_y = 0.5$. The FTCT parameters used where $\mathbf{n} = (9, 9)$ and $\tau = 2$. At the very last step we altered the function value at one of the grid points of a randomly chosen component solutions by a factor of $10^{-0.5}$ (as we described at the beginning of the section). Table 6.3 shows a list of all possible pairs $(\mathbf{s}, \mathbf{t})$ that can be generated from this combination technique, and for each pair we measured $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$. The pairs marked in boldface have an unusually large value, and inspecting this list should allow us to identify the corrupted solution. In this case, it is $u_{(7,8)}$, since it appears in all pairs marked as corrupted.

At this point there is still one question we haven't addressed: how do we tell if a given value of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ is "too large" (line 4 of Algorithm 6)? Is 10^2 too large? Or 10^{10}? This obviously depends on the problem under study, but we mentioned we were looking fo an algorithm that is not problem-dependent. One possible way to do this is to consider the value of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ for a given pair *relative* to the other pairs. In table 6.3 we see that the values of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ that were not affected by SDC are of the order of 10^{-2}. Relative to this value, 10^4 does seem disproportionately large. Our task is then not to tell if a value is larger than a given bound, but to determine if it is an outlier compared to the rest. For our preliminary tests in serial we forgo this discussion and leave the task of detecting outliers to existing Python libraries.

We now discuss a second possible way of implementing the function `SanityCheck`.

Algorithm 6: Sanity Check 1: Comparing Combination Solutions Pairwise via the Maximum Norm

```
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, $\mathcal{J}_{\text{sdc}}$
1 Function SanityCheck($\{u_i\}$)
2     for all pairs $(u_{\mathbf{s}}, u_{\mathbf{t}})$ with $\mathbf{s}, \mathbf{t} \in \mathcal{I}_{\tau}^{d,n}$ do
3         Compute $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ ;                                   // Eq. (6.11)
4         if $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ too large then
5             Mark pair $(\mathbf{s}, \mathbf{t})$ as corrupted;
6     From list of corrupted pairs $(\mathbf{s}, \mathbf{t})$, determine corrupted solutions;
7     return $\mathcal{J}_{sdc}$
```

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**

Figure 6.3: Measured values of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ for a 2D advection equation after SDC has been injected.

6.2.2 Sanity Check 2: Comparing Combination Solutions via their Function Values Directly

The idea of using outlier detection with the combination technique gives rise to an alternative way to detect if a solution has been affected by SDC. Consider the grid on which we combine all component solutions, $\Omega_{\mathbf{n}}^{(c)}$, which is a sparse grid. The function value $u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})$ at an arbitrary point $x_{\mathbf{l},\mathbf{j}}$ on this grid is obtained from the combination of all component solutions $u_{\mathbf{i}}$ that include grid point $x_{\mathbf{l},\mathbf{j}}$. For every grid point $x_{\mathbf{l},\mathbf{j}}$ there is always at least one component solution $u_{\mathbf{i}}$ that includes it (if the grid point belongs to any of the highest hierarchical subspaces, $W_{\mathbf{l}}$, $|\mathbf{l}|_1 = n + d \cdot (\tau + 1) - 1$) and at most $|\mathcal{I}|$ (if the grid point belongs to the lowest hierarchical subspace $W_{\mathbf{1}}$). We will denote $N_{\mathbf{l}} = 1, \ldots, |\mathcal{I}|$ the number of component solutions $u_{\mathbf{i}}$ that contain the grid points $x_{\mathbf{l},\mathbf{j}}$ corresponding to level $\mathbf{l}$. Naturally, we expect the values $u_{\mathbf{i}}(x_{\mathbf{l},\mathbf{j}})$ to be similar across the component solutions $u_{\mathbf{i}}$ that contain it, with some variation. The variance across the different $u_{\mathbf{i}}$ is given by

$$\mathrm{Var}[u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] = \frac{1}{N_{\mathbf{l}}} \sum_{\mathbf{l}' \geq \mathbf{l}} \left(u_{\mathbf{l}'}(x_{\mathbf{l},\mathbf{j}}) - \mathrm{E}[u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] \right)^2, \quad \mathbf{l}, \mathbf{l}' \in \mathcal{I}. \tag{6.12}$$

Note that a grid point $x_{\mathbf{l},\mathbf{j}}$ is contained in all component solutions $u_{\mathbf{l}'}$ with $\mathbf{l}' \geq \mathbf{l}$. For the mean value of $u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})$ over the solutions we have

$$\mathrm{E}[u_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] = \frac{1}{N_{\mathbf{l}}} \sum_{\mathbf{l}' \geq \mathbf{l}} u_{\mathbf{l}'}(x_{\mathbf{l},\mathbf{j}}). \tag{6.13}$$

Equivalently, in the hierarchical basis we have

$$\mathrm{Var}[\alpha_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] = \frac{1}{N_{\mathbf{l}}} \sum_{\mathbf{l}' \geq \mathbf{l}} \left(\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{l}')} - \mathrm{E}[\alpha_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] \right)^2. \tag{6.14}$$

It is straightforward to show using Eq. (6.6) that this variance is bounded by

$$\begin{aligned} \mathrm{Var}[\alpha_{\mathbf{n}}^{(c)}(x_{\mathbf{l},\mathbf{j}})] &= \frac{1}{2N_{\mathbf{l}}^2} \sum_{\mathbf{s} \geq \mathbf{l}} \sum_{\mathbf{t} \geq \mathbf{l}} \left(\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})} \right)^2 \\ &\leq \frac{8 \cdot \kappa^2(x_{\mathbf{l},\mathbf{j}})}{N_{\mathbf{l}}^2} \sum_{\mathbf{s} \geq \mathbf{l}} \sum_{\substack{\mathbf{t} \geq \mathbf{l} \\ \mathbf{t} \neq \mathbf{s}}} g^2(h_{\mathbf{s}}^p, h_{\mathbf{t}}^p). \end{aligned} \tag{6.15}$$

$$g(h_{\mathbf{s}}^p, h_{\mathbf{t}}^p) := \sum_{k=1}^{d} \sum_{\substack{\{e_1,\ldots,e_k\} \\ \subset \{1,\ldots,d\}}} \left(h_{s_{e_1}}^p \cdots h_{s_{e_k}}^p + h_{t_{e_1}}^p \cdots h_{t_{e_k}}^p \right) \tag{6.16}$$

In 2D, this expression reduces to $g(h_{\mathbf{s}}^p, h_{\mathbf{t}}^p) = h_{t_1}^p + h_{s_1}^p + h_{t_2}^p + h_{s_2}^p + h_{t_1}^p h_{t_2}^p + h_{s_1}^p h_{s_2}^p$.

Once again, we cannot use this bound for the variance to determine if a given value $\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{l}')}$ falls within a reasonable deviation, since it depends on the function κ. But once again we can test if there are any outliers compared to the rest of the values. The procedure is explained in Algorithm 7. The idea is to traverse each point in the sparse grid $\Omega_{\mathbf{n}}^{(c)}$ (line 2), and for each of them, gather all contributions from the different component solutions that contain the point (line 3). We then perform a simple outlier test on that grid point, and if any of the values does not pass the test, we mark the corresponding component solution as corrupted and add its index to $\mathcal{J}_{\mathrm{sdc}}$.

There are two main concerns with this approach. First, as we mentioned, many of the points on the sparse grid come from a single component solution $u_{\mathbf{i}}$, namely

Algorithm 7: Sanity Check 2: Comparing Combination Solutions via their Function Values Directly

```
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,
        $\mathcal{J}_\text{sdc}$
1 Function SanityCheck($\{u_\mathbf{i}\}$)
2     for all grid points $x_{\mathbf{l},\mathbf{j}}$ in $\Omega_\mathbf{n}^{(c)}$ do
3         $\alpha[\mathbf{l}'] \leftarrow$ gather($\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{l}')}$) for all $\mathbf{l}' \geq \mathbf{l}$;
4         if any outlier_test($\alpha[\mathbf{l}']$) then
5             Add outlier $\mathbf{l}'$ to set of corrupted indices $\mathcal{J}_\text{sdc}$;
6     return $\mathcal{J}_{sdc}$
```

those for which $|\mathbf{i}|_1 = n + d \cdot (\tau + 1) - 1$. Since it is not possible to perform an outlier test on a single value, this algorithm cannot be used for those grid points. The same applies if we have only a few values for a grid point. For example, if $|\mathbf{i}|_1 = n + d \cdot (\tau + 1) - 2$, we have at most $d + 1$ values to compare, which is not much for small d. Evidently, the more samples, the more robust the statistics. The minimum sample size depends on the algorithm used to detect outliers, and as we will see in the results, we usually need at least five samples.

So what can we do if we have fewer than, say, five samples? Unfortunately, not much. In some cases, if the function u we want to approximate is sufficiently smooth (such that the hierarchical surpluses always decay exponentially) we could check if the surpluses indeed decay as expected. In Fig. 6.4 we show the hierarchical surpluses corresponding to two similar-looking functions. On the left, all surpluses decay with increasing level $\mathbf{l}$. If the function is assumed to behave that way, one could check whether the surpluses on the hierarchically highest subspace are smaller than those on a lower subspace (for example, m levels lower):

$$|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{l})}| < |\alpha_{\mathbf{l}-m\cdot\mathbf{e}_k,\mathbf{j}}^{(\mathbf{l})}|, \tag{6.17}$$

where the direction $\mathbf{e}_k$ should preferably be chosen to be the one for which l_k is largest. But this might not always hold. On the right subfigure we show a function

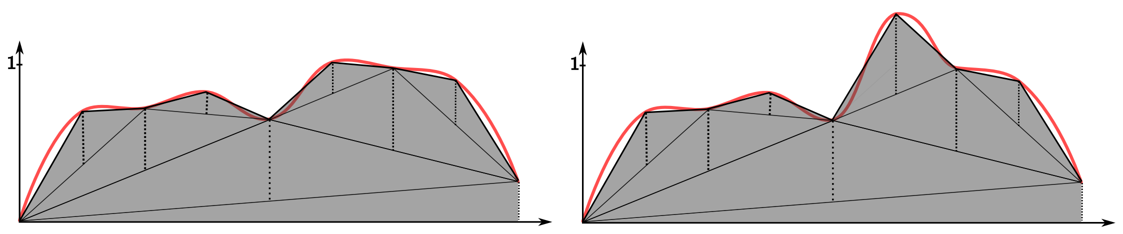


Figure 6.4: Left: Function with decaying surpluses. **Right**: Function where one surplus does not decay.

where not all surpluses decay as $\mathbf{l}$ increases. If this is the case, the test could give a false positive. As we will see in the results section, where we applied criterion (6.17), we did not have any false positives, but this is mostly due to the fact that the underlying problem is very well-behaved (a linear 2D advection equation). To visualize this, we solved the linear advection equation on grid $\Omega_{(8,7)}$ and measured the average of the absolute value of the surpluses over $\mathbf{j}$ for a fixed level $|\mathbf{l}|_1$,

$$\frac{1}{N} \sum_{\mathbf{j} \in \mathcal{I}_{\mathbf{l}}, |\mathbf{l}|_1 = c} \left| \alpha_{\mathbf{l},\mathbf{j}}^{(8,7)} \right| .$$

The values are plotted in Fig. 6.5 as a function of $|\mathbf{l}|_1$. We can see an almost perfect exponential decay.

6.2.3 Simulation Scenario: 2D Advection Equation

To test Algorithm 5 with our two sanity checks we chose to solve the advection equation (6.1) with $c_x = c_y = 1$ in Python. For the `solver` function we implemented a Lax-Wendroff scheme, which is of order two in space and time, so we set $p = 2$ for the error expansion. For the combination technique we used the parameters $\mathbf{n} = (9, 9)$ and $\boldsymbol{\tau} = (5, 5)$. The resulting combination scheme has ten grids in total (seven from the classical combination technique and three more to ensure fault tolerance). For the time discretization we simulated 512 time steps in the range $t \in [0, 0.5]$. The time step is the same for all combination solutions, and we combine the component solutions twice: after 256 time steps and at the end. We performed all simulations in serial – there is no parallelization involved at this point.

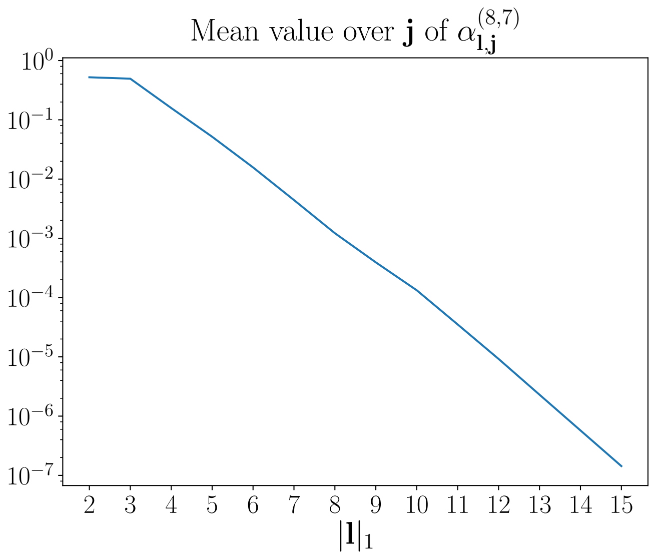


Figure 6.5: Decay of the hierarchical surpluses for the 2D linear advection equation.

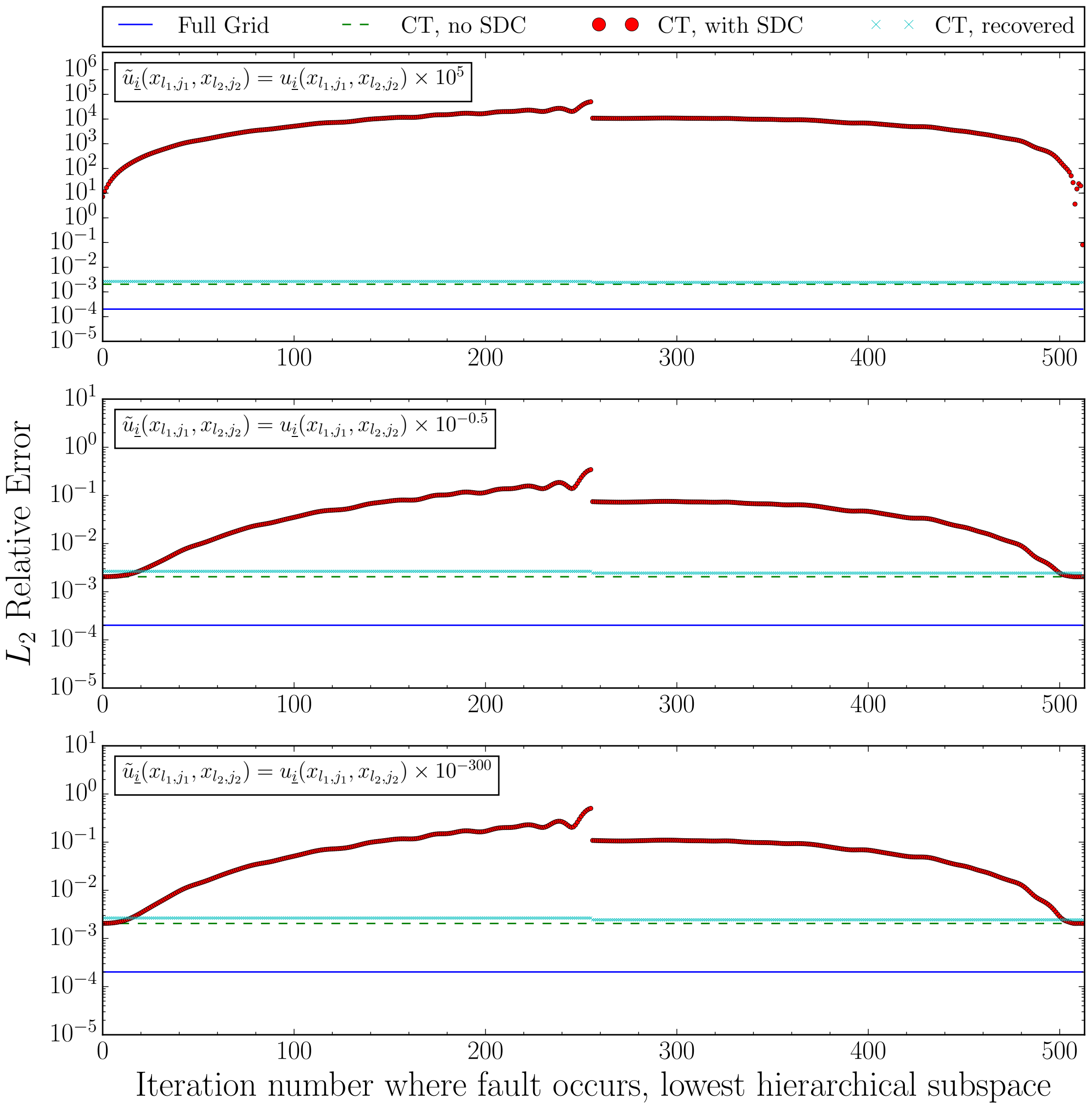


Figure 6.6: Error of the combination technique when SDC of various magnitudes is injected into the middle of the domain of one component solution[1].

We injected faults in the way described at the beginning of the chapter. We chose one of the seven component solutions randomly, but restricted our choice to the solutions with the highest discretization resolution, since these contain some of the hierarchical subspaces where SDC should be the hardest to detect (namely, those with the largest level $\mathbf{l}$). We then carried out 512 different simulations, injecting SDC once at iteration i for simulation i by altering the value either by a large factor (10^{+150}), a small factor ($10^{-0.5}$) or a factor close to zero (10^{-300}). Finally, the SDC was injected either at the middle of the 2D spatial domain (at coordinate $\vec{x} = (0.5, 0.5)$, corresponding to the hierarchically lowest subspace $W_{(0,0)}$), or near the middle of the domain (at coordinate $\vec{x} = (0.5 - h_1, 0.5 - h_2)$, corresponding to the hierarchically highest subspace $W_{\mathbf{l}}$ with $|\mathbf{l}|_1 = 15$). This gives a total of $512 \times 3 \times 3 = 3,072$ simulations.

For detecting outliers we used the Python library *statsmodels* [SP10]. In particular, we used the function `outlier_test` found in the module `linear_model` which, in version 0.7.0, implements seven different outlier detection algorithms, all of which we observed to perform quite similarly for our experiments. We chose the option `method='fdr_by'` which implements the false discovery rate (FDR) method described in [BY01].

6.2.4 Results

Sanity Check 1: Comparing Solutions Pairwise via the Maximum Norm

After testing our implementation of this sanity check extensively, we were not able to obtain satisfactory results. As we will see in the next section, a robust algorithm to detect outliers requires some information about where the data came from and what can be reasonably expected. As it turns out, the values of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ behave quite erratically and it is therefore not straightforward to detect outliers in the measurements. We will see how to solve this problem in the next section, so we postpone this discussion until then.

Sanity Check 2: Comparing Function Values Directly

Our second sanity check does not face the same problem described above: we run the outlier detection algorithm on constant values, which is the simplest scenario one can have.

Our results are summarized in Figs. 6.6 and 6.7. Figure 6.6 shows the error of the combination technique when SDC is injected on the middle of the domain. The three subfigures correspond to the three different orders of magnitude of SDC described earlier[2] On the x-axis we see the iteration number at which SDC was injected, and on the y-axis the relative L_2 error of the combination technique measured at the end of the 512 time steps compared to the exact solution of the PDE. What we observe is that SDC was detected in all but one case[3], since the light blue crosses (CT after recovery) and the red circles (CT with SDC) do not overlap. Recovery means that the SDC was detected and alternative combination coefficients were used to exclude the wrong component solution. As with hard faults, we see that the error of the fault tolerant combination technique is very close to that of the combination technique without faults.

Another interesting observation that can be read off the second and third sets of results is that the outlier detection algorithm seems to be very sensitive to any variations. If we observe what happens when SDC is injected during the first few

[1]Figure as published in [PHHHB16].

[2]The largest magnitude, 10^{+150}, was substituted by 10^5 simply to visualize the plots better.

[3]SDC was not detected when it was injected at the last iteration with magnitude $10^{-0.5}$, possibly because at this iteration, the value of the solution is very close to zero.

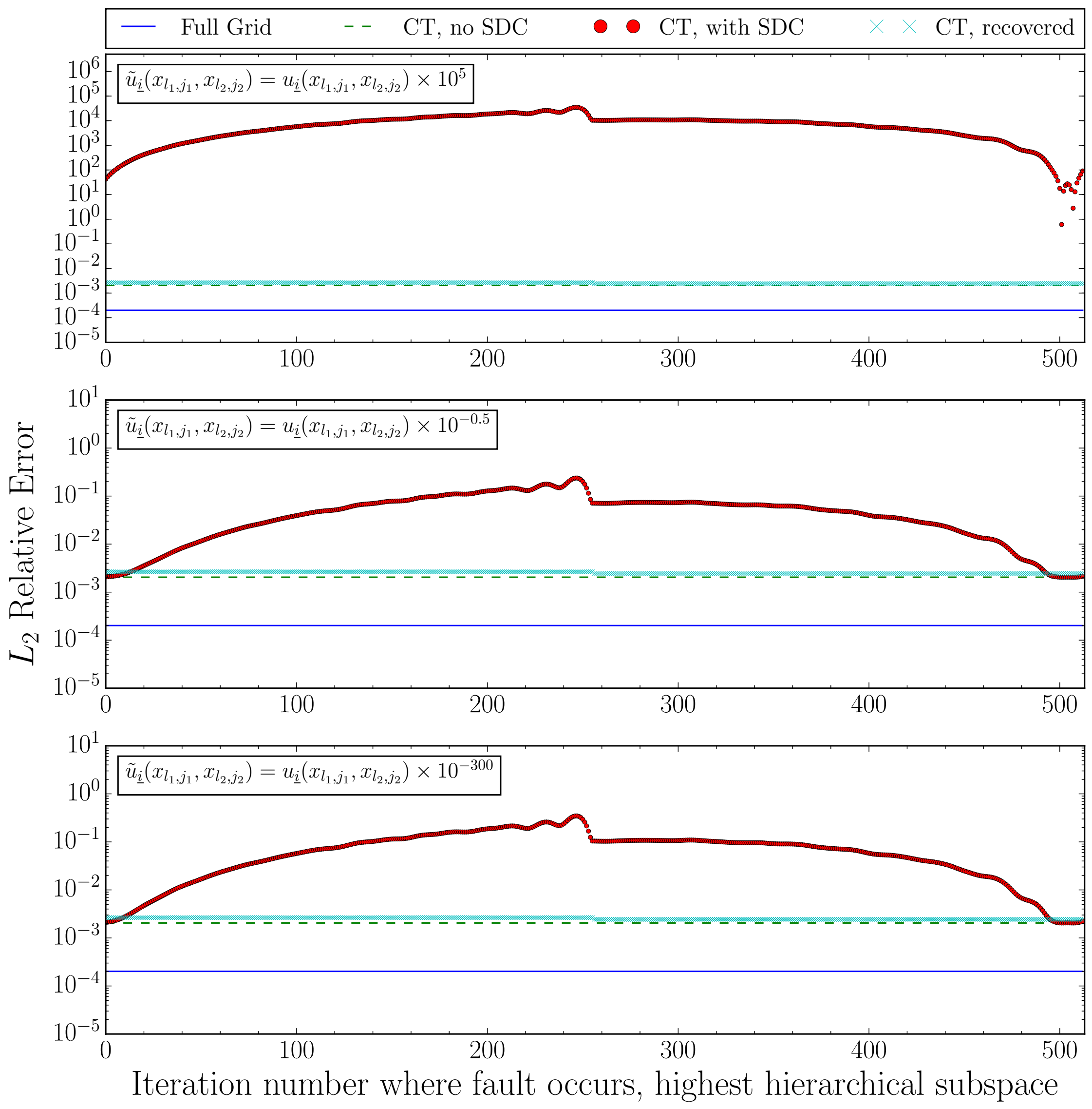


Figure 6.7: Error of the combination technique when SDC of various magnitudes is injected near the middle of the domain of one component solution.

or the last few iterations, we notice that the error of the combination technique with SDC is *smaller* than after recovery. In other words, in those cases it would have been preferable *not* to detect the SDC, but the difference is so small that we can afford this extra sensitivity.

Figure 6.7 shows the same sets of simulations but with SDC injected near the middle of the domain (at the grid point $\vec{x} = (0.5 - h_1, 0.5 - h_2)$, corresponding to the hierarchically highest subspace). Here we were also able to detect SDC in all cases, but also due to our additional check (6.17). As we discussed earlier, this might not always work, or it might return false positives. We will talk about this in the next section in more detail.

Finally, Fig. 6.8 shows the same sets of simulations as Fig. 6.6 but here we

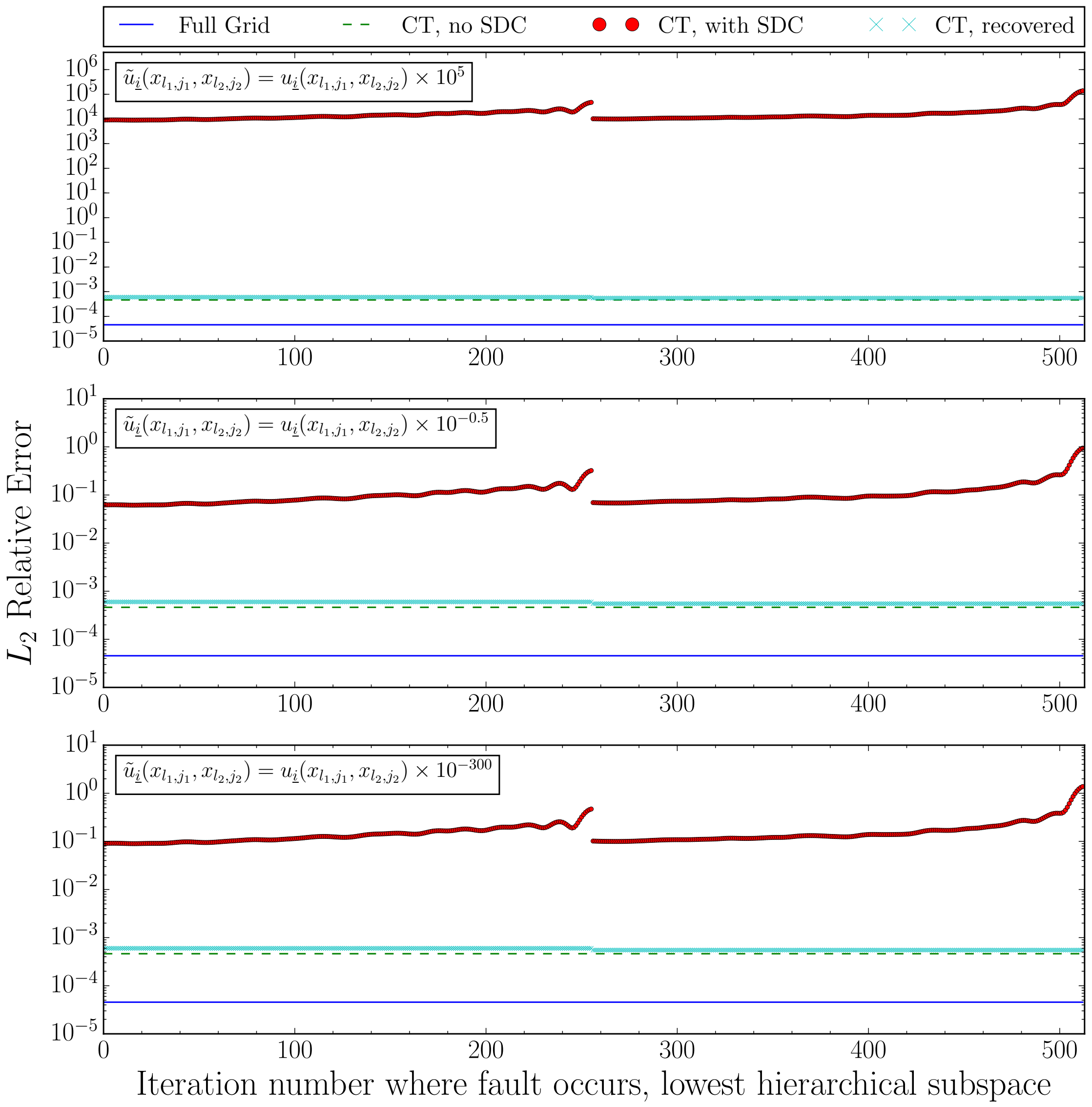


Figure 6.8: Same sets of simulations as Fig. 6.6 but adding a constant term to the solution of the PDE.

added a constant term π to the exact solution of the PDE,

$$u(x, y, t) = \sin(2\pi(x - c_x t)) \sin(2\pi(y - c_y t)) + \pi.$$

We do this so that the function values are never close to zero and we can exclude those cases from our tests. The outlier detection works just as well as in the previous sets of simulations, but as expected, the error with SDC is larger.

In sets of simulations with smaller truncation parameters $\boldsymbol{\tau}$, the outlier detection didn't work as well. There were both false positives and false negatives. But this was not a problem of the detection algorithm, but of the combination technique itself. If the truncation parameter is too small and we thus have strongly anisotropic grids, the overall quality of the combination technique is noticeably damaged. We observed that, as long as $\boldsymbol{\tau}$ is chosen large enough to result in a good classical combination, the outlier detection algorithms will work fine.

These preliminary results motivated us to investigate two more questions. First, we believe that the first sanity check could still be useful for cases where SDC affects the hierarchically highest subspaces, but we need to reformulate it. And second, our algorithms should be able to perform well in parallel, but at this point we haven't made any attempts to make our algorithms efficient and scalable. The answers to these two questions are the topic of the next section.

6.3 Large-Scale Experiments in Parallel

We are still several steps away from a robust and efficient implementation of a silent error detection algorithm. As we saw in the previous section, it is not enough to hope that an outlier detection algorithm can reliably tell us which observations of $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ are wrong without specifying more information about the underlying numerics. This is the basic requirement to make the first sanity check work. Additionally, it is not clear whether our second sanity check would scale in a parallel setting, since we run the outlier detection algorithm once for each of the sparse grid points. Our aim for this section is to solve these problems and put them to test on a massively parallel scenario.

6.3.1 Detecting Outliers via Robust Regression

Since both sanity checks described rely on the detection of outliers, and given that one detection algorithm failed after a naive implementation, we turn our attention to the actual problem of detecting outliers, an old problem in statistics. Many methods have been developed over the years, and for our type of problem we believe *robust regression* offers just the right framework. Many books have been written on the topic of robust regression, and the following discussion is based on the excellent book by Rousseeuw and Leroy [RL05].

Consider the set of measurements shown in Fig. 6.9a. If asked whether there are any outliers among the measurements, we could be tempted to answer "yes" – the eleventh data point doesn't seem to follow the same trend as the rest (a roughly linear increase). But now assume we take more measurements and obtain Fig. 6.9b. It is not clear anymore that there are outlier measurements. In fact, the dataset was generated in a way such that it follows the function shown in Fig. 6.9c with some random noise, but no outliers. Indeed, if we calculate the residuals $y_i - y(x_i)$ we see that they are all in the same order of magnitude, as can be seen in Fig. 6.9d.

This is the main idea of using regression to find outliers. The measurements themselves are not enough to tell whether there are outliers: if we know the underlying mathematical model behind the measurements, we should look at the *residuals* after fitting the model to the measurements. If one or more residuals are too large, then they can act as indicators for outliers.

In our example, imagine we know that the underlying model has the general

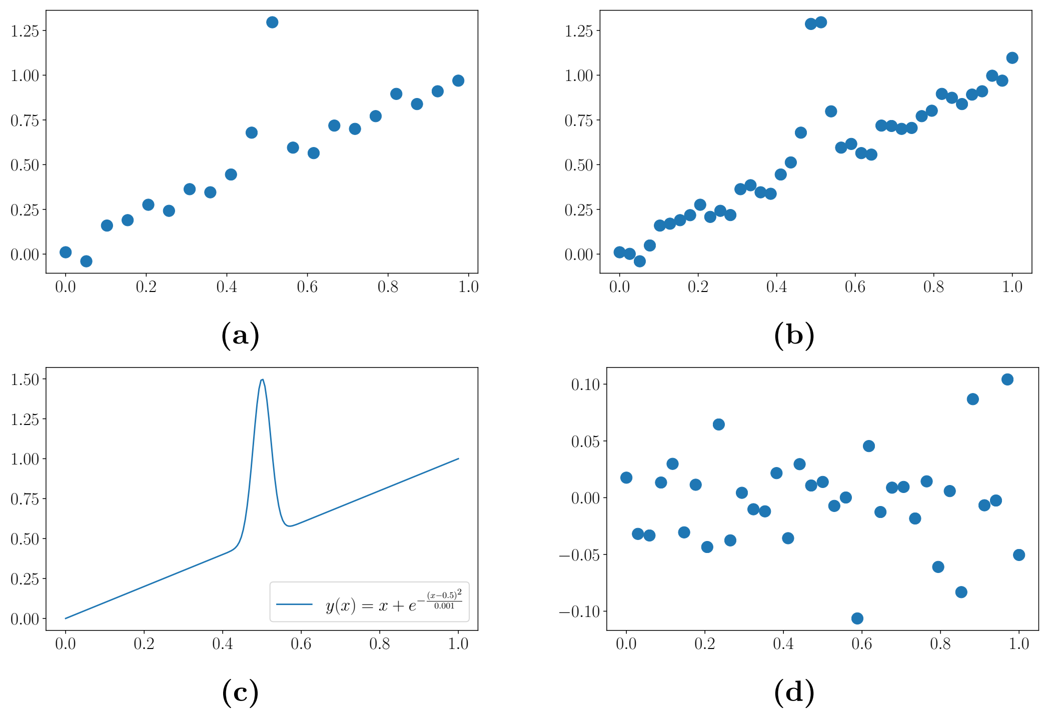


Figure 6.9: **(a)** 20 measurements of an experiment; **(b)** Same experiment as **(a)** but with 20 more measurements; **(c)** Underlying function for the previous measurements; **(d)** Residuals $y_i - y(x_i)$.

form

$$y(x) = c_0 x + c_1 e^{-\frac{(x-c_2)^2}{c_3}}. \tag{6.18}$$

In this case, our task would be to find the constants c_0, c_1, c_2 and c_3 that fit our model best, which can be easily done with simple least squares regression. But things get complicated if there are outliers in the data. Simply applying least squares to the data is not enough, since the minimization problem would try to fit the outliers as well, returning wrong results. Going back to our example, this means that the constants c_0 to c_3 should have roughly the same values in the absence *and* in the presence of outliers, such that the residuals are meaningful. This is where we enter the domain of *robust regression* – fitting models to data in the presence of outliers.

The simplest example where simple regression fails is when fitting a constant data that we expect to be roughly constant, for example

$$3.35, \quad 3.42, \quad 3.45, \quad \mathbf{21.20}, \quad 3.64, \quad 3.54, \quad 3.56, \quad 3.39$$

The least squares problem

$$c_{\min} \leftarrow \min_c \frac{1}{2} \sum_{i=1}^{8} (y_i - c)^2 \tag{6.19}$$

is solved by choosing c to be the mean value over the measurements,

$$c_{\min} = \frac{1}{8}\sum_{i=1}^{8} y_i. \tag{6.20}$$

For our measurements, we obtain $c_{\min} = 5.69$, which is clearly far off from the real answer 3.5 due to the presence of the outlier $y_4 = 21.20$. In short, the least squares estimator (6.19) is not robust to outliers. One could instead take the l_1 estimator given by

$$c_{\min} \leftarrow \min_c \sum_{i=1}^{8} |y_i - c|, \tag{6.21}$$

whose solution is the *median* of the measurements, in our case 3.49. The downside is that solving problem (6.21) is harder than solving the simple least squares problem, since the absolute value function is not everywhere differentiable [sci].

One way to overcome this complication is by substituting the l_1 estimator by a smoother function, usually a *sublinear* function ρ (meaning that it grows slower than linearly) [sci],

$$c_{\min} \leftarrow \min_c \sum_{i=1}^{N} \rho\left(y_i - y(c, x_i)\right). \tag{6.22}$$

ρ is our new loss function, and it should fulfill the following requirements [GDT+15]:

$$\begin{aligned} &\rho(e) \geq 0 \\ &\rho(0) = 0 \\ &\rho(-e) = \rho(e) \\ &\rho(e_1) \geq \rho(e_2) \quad \text{for} \quad |e_1| > |e_2| \end{aligned} \tag{6.23}$$

Choosing $\rho(e) = e^2$ is equivalent to the ordinary least squares problem. Some common choices in robust regression include

- Huber's function:
$$\rho(e) = \begin{cases} e^2, & e \leq 1 \\ e - 1, & e > 1 \end{cases}$$
- Cauchy's function:
$$\rho(e) = \ln(1 + e^2)$$
- arctan function:
$$\rho(e) = \arctan(e^2).$$

By now there are many well-known algorithms to solve the robust minimization problem (6.22), most noticeably the Trust Region Reflective algorithm [BCL95] or the Iteratively Reweighted Least Squares (IRLS) method [HW77].

With these ideas in mind, we can revisit our two sanity checks and see how this theoretical framework applies to them.

6.3.2 Sanity Check 1 Revisited

The original idea consisted in measuring the largest normalized difference of pairs of component solutions,

$$\hat{\beta}^{(\mathbf{s},\mathbf{t})} := \max_{\mathbf{l}\leq \mathbf{s}\wedge\mathbf{t}} \max_{\mathbf{j}\in\mathcal{I}_{\mathbf{l}}} \frac{|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|}{\min\left\{|\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}|, |\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}|\right\}} \quad \text{for all } \mathbf{l}\leq\mathbf{s}\wedge\mathbf{t}, \quad \mathbf{0}\leq\mathbf{j}\leq 2^{\mathbf{l}} \tag{6.24}$$

If we now search for outliers in these measurements, we should have an idea of what the underlying mathematical model for $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$ is. In fact, we do have a model for the numerator, namely the error splitting assumption,

$$\begin{aligned}\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})} = \sum_{k=1}^{d} \sum_{\substack{\{e_1,\dots,e_k\}\\ \subset\{1,\dots,d\}}} \Big(& D_{e_1,\dots,e_k}(\vec{x}, h_{t_{e_1}},\dots,h_{t_{e_k}}) h_{t_{e_1}}^p \cdots h_{t_{e_k}}^p \\ & - D_{e_1,\dots,e_k}(\vec{x}, h_{s_{e_1}},\dots,h_{s_{e_k}}) h_{s_{e_1}}^p \cdots h_{s_{e_k}}^p \Big).\end{aligned} \tag{6.25}$$

But unfortunately we do not have a model for the normalized absolute value, so it seems that Eq. (6.24) is not a good candidate to apply robust regression to.

The good news is that maybe we do not need to take the absolute value and normalize it. Recall that the original motivation behind the normalization was to compensate for the exponential decay across the levels $\mathbf{l}$. But if our model already accounts for this (through the functions $D_{\mathbf{i}}$) then it might not be necessary to normalize.

Let us describe the procedure in two dimensions. Suppose we have many measurements of $\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})}$ - $\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})}$ for a list of pairs $(\mathbf{s},\mathbf{t})$. We would then try to fit these measurements to the model

$$\begin{aligned}\alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{t})} - \alpha_{\mathbf{l},\mathbf{j}}^{(\mathbf{s})} &= D_1(x_{\mathbf{l},\mathbf{j}}, h_{t_1}) h_{t_1}^p + D_2(x_{\mathbf{l},\mathbf{j}}, h_{t_2}) h_{t_2}^p + D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{t_1}, h_{t_2}) h_{t_1}^p h_{t_2}^p \\ &- D_1(x_{\mathbf{l},\mathbf{j}}, h_{s_1}) h_{s_1}^p - D_2(x_{\mathbf{l},\mathbf{j}}, h_{s_2}) h_{s_2}^p - D_{1,2}(x_{\mathbf{l},\mathbf{j}}, h_{s_1}, h_{s_2}) h_{s_1}^p h_{s_2}^p,\end{aligned} \tag{6.26}$$

which would mean finding the values of the three functions $D_{\mathbf{i}}$ at the grid points $x_{\mathbf{l},\mathbf{j}}$ and mesh sizes h_i. These are *many* unknowns to fit and therefore makes the regression problem unfeasible, but the problem can be greatly simplified if we focus only on one grid point – the same grid point over all pairs. We will be interested only on the sparse grid point $x_{\mathbf{l},\mathbf{j}}^{*}$ for which the absolute value of $\beta^{(\mathbf{s},\mathbf{t})}$ is largest, since this point should be our first suspect in the search for SDC:

$$(\mathbf{s},\mathbf{t})^* = \arg\max_{(\mathbf{s},\mathbf{t})\in\mathcal{V}} \left|\beta^{(\mathbf{s},\mathbf{t})}\right| \tag{6.27}$$

$$x_{\mathbf{l},\mathbf{j}}^{*} = \arg\max_{x_{\mathbf{l},\mathbf{j}}} \left|\beta_{\mathbf{l},\mathbf{j}}^{(\mathbf{s},\mathbf{t})^*}\right| \tag{6.28}$$

where $\mathcal{V}$ is the set of all pairs of multi-indices $(\mathbf{s},\mathbf{t})$ we are considering. In other words, we first search for the pair of component solutions for which $\beta^{(\mathbf{s},\mathbf{t})}$ is largest and then, for that pair, we store the grid point $x^*_{\mathbf{l},\mathbf{j}}$ that maximizes $\beta^{(\mathbf{s},\mathbf{t})}_{\mathbf{l},\mathbf{j}}$. By looking at only this sparse grid point in all pairs, we get rid off the spatial dependence of the functions $D_{\mathbf{i}}$. If the sparse grid point $x^*_{\mathbf{l},\mathbf{j}}$ does not exist on a component grid $u_{\mathbf{i}}$, we assign a value of zero to the hierarchical coefficient, $\alpha^{(\mathbf{i})}_{\mathbf{l},\mathbf{j}} = 0$.

As a final simplification we can get rid of the higher order h_i^p terms in the expansion (6.24),

$$\begin{aligned}\beta^{(\mathbf{s},\mathbf{t})} &:= \alpha^{(\mathbf{t})}(x^*_{\mathbf{l},\mathbf{j}}) - \alpha^{(\mathbf{s})}(x^*_{\mathbf{l},\mathbf{j}}) \\ &\approx \sum_{k=1}^{d} \left(D_{e_k}(x^*_{\mathbf{l},\mathbf{j}}, h_{t_{e_k}}) h^p_{t_{e_k}} - D_{e_k}(x^*_{\mathbf{l},\mathbf{j}}, h_{s_{e_k}}) h^p_{s_{e_k}} \right) =: \tilde{\beta}^{(\mathbf{s},\mathbf{t})}.\end{aligned} \tag{6.29}$$

Here, $\beta^{(\mathbf{s},\mathbf{t})}$ is the actual measurement of the surpluses and $\tilde{\beta}^{(\mathbf{s},\mathbf{t})}$ is the theoretical model we'll use for the robust fit. In 2D this would mean finding the values of the functions C_1 and C_2 evaluated at the mesh sizes h_i appearing in the combination technique, $h_i = \{h_{\tau+1}, h_{\tau+2}, \dots, h_n\}$[4]. Instead of making this measurement for all possible pairs of component solutions, we propose to do it only for pairs $(\mathbf{s},\mathbf{t}) \in \mathcal{V}$ that are nearest neighbors (in the sense that $|\mathbf{s}-\mathbf{t}|_1$ is smallest). Of course, the more grid points two solutions $u_{\mathbf{t}}$ and $u_{\mathbf{s}}$ have in common, the better. If a pair of solutions has only a few grid points in common, we might miss any SDC that affects the grid points that they don't have in common. This means that in d dimensions we should compare each solution with its d nearest neighbors.

Here's a 2D example with $n = 4$ and $\boldsymbol{\tau} = (\tau, \tau) = (3, 3)$, resulting in the following 10 multi-indices for the combination technique:

$$\bigcup_{q=0}^{3} \mathcal{I}^{2,4}_{q,(3,3)} = \{(7,4),(6,5),(5,6),(4,7),(6,4),(5,5),(4,6),(5,4),(4,5),(4,4)\} \tag{6.30}$$

Choosing to compare only the two nearest neighbors for each solution results in a set $\mathcal{V}$ with 11 pairs, listed in Table 6.1 along with measurements of $\beta^{(\mathbf{s},\mathbf{t})}$ for a simple example advection-diffusion equation (which will be described in more details in Sec. 6.3.5). SDC was injected into solution $u_{(4,6)}$, and as we can see from the table, the values corresponding to $\{(4,7),(4,6)\}$ and $\{(5,6),(4,6)\}$ are somewhat higher than the rest. Notice that the difference in orders of magnitude is not as large as with the normalized $\hat{\beta}^{(\mathbf{s},\mathbf{t})}$, but this is not a problem. The underlying model for the error expansion $\tilde{\beta}^{(\mathbf{s},\mathbf{t})}$ should be able to identify these outliers.

The minimization problem that results from this model is

$$\vec{c}_{\min} \leftarrow \min_{\vec{c}} \sum_{(\mathbf{s},\mathbf{t})\in\mathcal{V}} \rho\left(\beta^{(\mathbf{s},\mathbf{t})} - \tilde{\beta}^{(\mathbf{s},\mathbf{t})}(\vec{c})\right), \tag{6.31}$$

where $\vec{c}$ denotes the vector of unknown functions D_i,

$$\vec{c} := (D_1(h_{\tau+1}), \dots, D_1(h_n), \dots, D_d(h_{\tau+1}), \dots, D_d(h_n)).$$

[4]To keep the notation simple, we will only consider constant truncation parameters $\boldsymbol{\tau} \equiv \tau \cdot \mathbf{1}$.

Pair $(\mathbf{s}, \mathbf{t})$	$\beta^{(\mathbf{s},\mathbf{t})}$
(4, 5) (4, 4)	0.0275
(4, 7) (4, 6)	**0.2180**
(4, 7) (5, 6)	-0.0029
(5, 4) (4, 4)	0.0152
(5, 4) (5, 6)	-0.0498
(5, 5) (4, 5)	0.0158
(5, 6) (4, 6)	**0.2210**
(6, 5) (5, 5)	0.0111
(6, 5) (6, 4)	0.0283
(7, 4) (6, 4)	0.0061
(7, 4) (6, 5)	-0.0222

Table 6.1: Measurements of $\beta^{(\mathbf{s},\mathbf{t})}$ with one solution affected by SDC, namely, $u_{(4,6)}$.

This is a vector of $d \cdot (n - \tau)$ unknowns. Notice also that our model $\tilde{\beta}^{(\mathbf{s},\mathbf{t})}$ is linear in the unknowns D_i, so it can be written as a matrix-vector product

$$\tilde{\beta}^{(\mathbf{s},\mathbf{t})}(\vec{c}) := \mathbf{X} \cdot \vec{c},$$

where the matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d \cdot (n-\tau)}$ stores the coefficients h_i^p.

We can now solve the optimization problem (6.31) using any algorithm for robust linear regression. We opted for the implementation of the Iteratively Reweighted Least Squares (IRLS) algorithm found in the GSL library [GDT+15].

Iteratively Reweighted Least Squares

The algorithm works as follows. In order to solve

$$\min_{\vec{c}} \sum_i \rho(e_i(\vec{c}))$$

we take the derivative of ρ with respect to e and set the expression to zero,

$$\sum_i \psi(e_i)\mathbf{X}_i = 0,$$

where $\psi = \rho'$ and $\mathbf{X}_i$ is the i-th row of $\mathbf{X} \in \mathbb{R}^{M \times N}$. We then introduce a weight function $w(e) = \psi(e)/e$, which gives

$$\sum_i w_i e_i \mathbf{X}_i = 0,$$

where $w_i = w(e_i)$. For example, the weight function w corresponding to the Cauchy loss function would be

$$w = \frac{1}{1 + e^2}.$$

This is similar to minimizing a weighted least squares problem, but the weights w_i depend on the residuals e_i, which in turn depend on the minimization coefficients $\vec{c}$, which themselves depend on the weights. This means that the problem has to be solved iteratively, and the idea is to assign smaller weights to outlier measurements on each iteration. The IRLS does this as follows:

1. Obtain an initial guess $\vec{c}^{(0)}$ for the coefficients $\vec{c}$ using ordinary least squares.
2. At iteration k compute the residuals
   $$e_i^{(k)} = \frac{(y_i - \mathbf{X}_i \cdot \vec{c}^{(k-1)})}{t \cdot \bar{\sigma}^{(k)}\sqrt{1 - h_{i,i}}},$$
   where $\bar{\sigma}$ is an approximation of the standard deviation of the residuals, defined as $\bar{\sigma} = \text{MAD}/0.675$, with MAD being the Median-Absolute-Deviation of the $M - N$ largest residuals of iteration $k - 1$. The elements $h_{i,i}$ are called *statistical leverages* and they are the diagonal elements of the *projection matrix* $\mathbf{H} = \mathbf{X} \cdot (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$. The parameter t is a tuning constant that depends on the weight function w.
3. Update weights as $w_i^{(k)} = \psi(e_i^{(j)})/e_i^{(k)}$.
4. Solve the weighted least squares problem with the new weights to find the new coefficients $\vec{c}^{(k)}$.
5. Iterate steps 2 through 4 until a given convergence criterion for the coefficient vector is met.

It is also possible to scale the weight function w in order to penalize outliers more strongly. This can be done by introducing a constant C such that

$$\hat{w}(e^2) = C^2 w\left(\left(\frac{e}{C}\right)^2\right).$$

If $C < 1$, outliers are penalized more strongly, meaning they are assigned smaller weights at every iteration.

Figure 6.10 illustrates what the algorithm does. On the top figure we observe some measurements of $\beta^{(\mathbf{s},\mathbf{t})}$ without outliers (blue circles). In this case, also a simple (non-robust) regression algorithm will fit the model to the measurements well (green line). In the middle figure we added some outliers, and we see that the model cannot be fitted well if we use the same regression algorithm as without outliers. The fit is ruined by the presence of outliers. On the bottom figure we used the IRLS algorithm, and the outliers are properly excluded from the fit. One can easily see that the residuals

$$\vec{r} = \beta^{(\mathbf{s},\mathbf{t})} - \mathbf{X} \cdot \vec{c}_{\min} \tag{6.32}$$

(difference between the blue circles and the green line) are large for these two measurements.

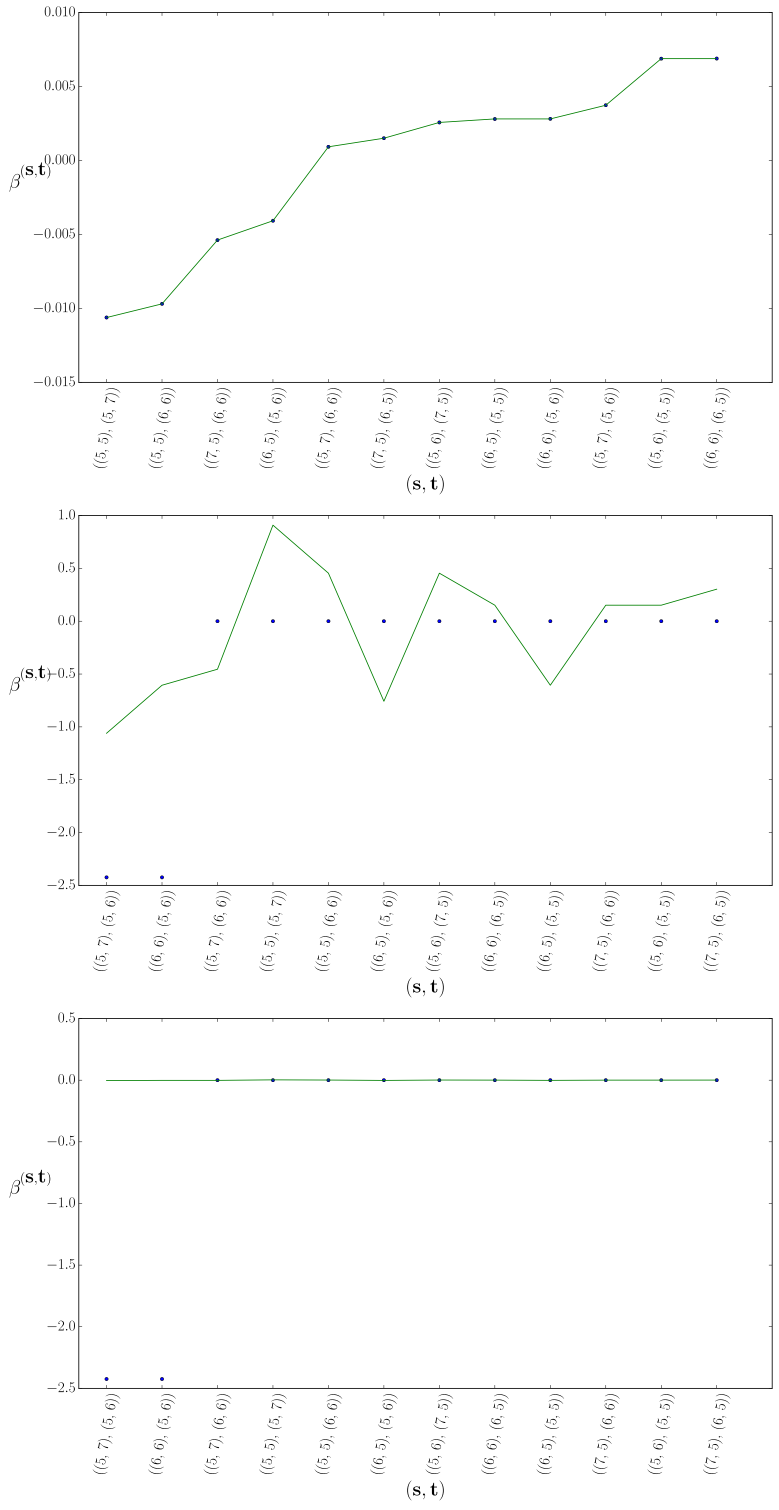


Figure 6.10: Top: $\beta^{(\mathbf{s},\mathbf{t})}$ measurements without outliers (blue circles) and the resulting least squares fit (green line); **Middle:** $\beta^{(\mathbf{s},\mathbf{t})}$ measurements with two outliers. A simple least squares fit is ruined by the presence of the outliers. **Bottom:** Robust least squares fit obtained with ILRS. Outliers are properly excluded.

Once the algorithm converges, we have the vector of robust coefficients $\vec{c}_{\min}$ and the corresponding residuals $\vec{r}$. The only thing left to define is a threshold for the residuals beyond which a measurement is marked as outlier, and this threshold should be scale-independent. One possible way to do this is described in [RL05]. First, one can calculate a preliminary scale estimate σ^0 defined as

$$\sigma^0 = 1.4826\left(1 + \frac{5}{|\mathcal{V}| - d \cdot (n-\tau)}\right)\sqrt{\text{med } \vec{r} \cdot \vec{r}}. \tag{6.33}$$

Here, $|\mathcal{V}|$ is the number of pairs $(\mathbf{s}, \mathbf{t})$, which is also the total number of $\beta^{(\mathbf{s},\mathbf{t})}$ measurements we have; $d \cdot (n-\tau)$ is the number of unknown D_i functions. For each residual, one then calculates a weight w_i given by

$$w_i = \begin{cases} 1, & \text{if} \quad |r_i/\sigma^0| \leq 2.5 \\ 0, & \text{otherwise} \end{cases}$$

After computing these weights one calculates a more robust scale estimate σ^*, which is given by

$$\sigma^* = \sqrt{\frac{\sum_{i=1}^{|\mathcal{V}|} w_i r_i^2}{\sum_{i=1}^{|\mathcal{V}|} w_i - d \cdot (n-\tau)}}.$$

Having this more robust scale estimate, one can compute the *standardized residuals* defined as

$$\vec{\hat{r}} = \frac{\vec{r}}{\sigma^*}. \tag{6.34}$$

It is common practice is to mark the i-th measurement as outlier if $|\hat{r}_i| > 2.5$.

Costs

The algorithm requires two main steps: 1) searching for the sparse grid point $x^*_{\mathbf{l},\mathbf{j}}$ for which $\beta^{(\mathbf{s},\mathbf{t})}$ is largest over all pairs of component solutions and 2) solving the robust least squares problem (6.31).

In order to find $x^*_{\mathbf{l},\mathbf{j}}$, we should calculate $\beta^{(\mathbf{s},\mathbf{t})}$ for all pairs of solutions. There are

$$|\mathcal{I}^{d,n}_{\boldsymbol{\tau}}| = \sum_{q=0}^{d-1} \frac{(n + (d-2) - q - \tau)!}{(n - q - \tau - 1)!(d-1)!} \tag{6.35}$$

component solutions in the truncated combination technique, as can be verified using, for example, the stars and bars method. For large n we have

$$\lim_{n\to\infty} |\mathcal{I}^{d,n}_{\boldsymbol{\tau}}| = \mathcal{O}(d \cdot n^{d-1}).$$

Comparing each component solution to its d nearest neighbors as we suggested results in $|\mathcal{V}| = \mathcal{O}(d^2 \cdot n^{d-1})$ total pairs, so we only have an additional d factor.

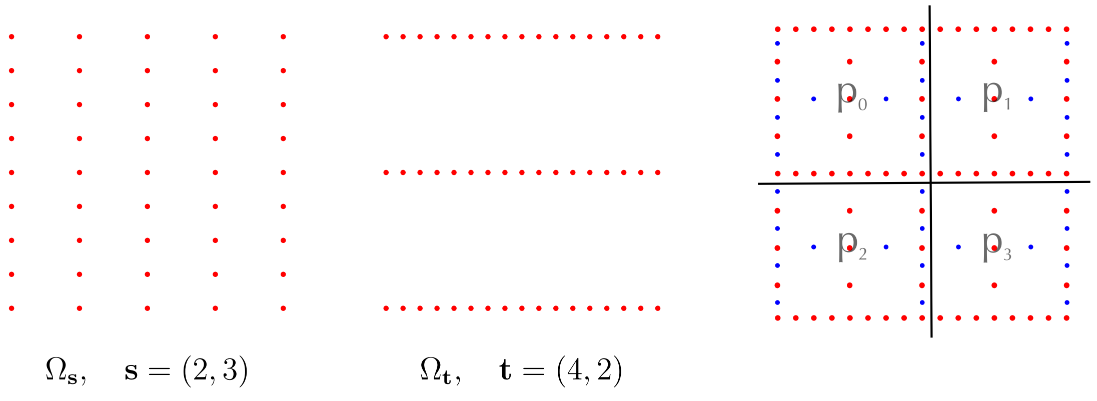

$\Omega_{\mathbf{s}}, \quad \mathbf{s} = (2,3)$
$\Omega_{\mathbf{t}}, \quad \mathbf{t} = (4,2)$
p_0
p_1
p_2
p_3

Figure 6.11: Two component grids can be superposed on (somewhat of) a sparse grid space, where the search for $\beta^{(\mathbf{s},\mathbf{t})}$ can be parallelized.

Each component solution has $\mathcal{O}(2^n)$ grid points, and computing $\beta^{(\mathbf{s},\mathbf{t})}$ for one pair requires simply traversing both solutions and subtracting one from the other, which is basically for free. Therefore, computing $\beta^{(\mathbf{s},\mathbf{t})}$ for all pairs should not represent a high cost.

Solving minimization problem (6.31) is also quite inexpensive with IRLS. The model matrix $\mathbf{X}$ has size $|\mathcal{V}| \times d \cdot (n - \tau)$, which is very small even for large n and d. The IRLS takes roughly $10^1 - 10^2$ iterations to converge (as observed in our experiments), so the cost of the minimization algorithm is negligible (which we confirmed in our experiments).

Parallelization

The algorithm can be easily parallelized using our software framework for the combination technique. As we know, the component solutions are distributed among the process groups, and each solution is parallelized using domain decomposition. This allows us to compute $\beta^{(\mathbf{s},\mathbf{t})}$ for each pair in a distributed manner, as shown in Fig. 6.11. We perform the operation $|\alpha^{(\mathbf{t})}_{\mathbf{l},\mathbf{j}} - \alpha^{(\mathbf{s})}_{\mathbf{l},\mathbf{j}}|$ on the distributed sparse grid, so each process computes its local value for $\beta^{(\mathbf{s},\mathbf{t})}$ and we then use `MPI_Reduce` to find the largest value. Once we have a list of $\beta^{(\mathbf{s},\mathbf{t})}$ values for all pairs in one group (that is, $\mathbf{s}, \mathbf{t} \in \mathcal{I}_g$), the process that contains the grid point with the largest value computes $\beta^{(\mathbf{s},\mathbf{t})}(x^*_{\mathbf{l},\mathbf{j}})$ for all pairs and performs the robust regression. Each process group does this in parallel before the combination step. It would also be possible to gather all values of $\beta^{(\mathbf{s},\mathbf{t})}$ across all groups in order to have more measurements, but this would require global communication, and we do not consider it necessary. As long as each group has enough component solutions for there to be enough measurements, we can do everything at the group level. As we will see in our experiments, this is almost always the case, especially if we run large scale simulations, since they usually result in dozens of component grids per group. The equations appearing in this section should simply be updated to contain $\mathcal{I}_g$ and $\mathcal{V}_g$ instead of $\mathcal{I}$ and $\mathcal{V}$.

6.3.3 Sanity Check 2 Revisited

Recall the main idea of our second approach: observing each sparse grid point individually, gathering all versions of the solution on that point and trying to detect any outliers across all component solutions. As we discussed, this algorithm had two main drawbacks. First, we performed outlier detection on *all* sparse grid points, which is quite expensive, especially as the dimension increases. Second, we didn't have a very robust way to detect outliers for the sparse grid points on the hierarchically highest subspaces, since we might only one (or very few) versions of the PDE solution for those points. We now try to address these two issues.

In order to avoid performing the outlier detection on each sparse grid point, we can repeat the first step of the algorithm corresponding to our first sanity check, namely, finding the *one* grid point $x^*_{\mathbf{l},\mathbf{j}}$ for which $\beta^{(\mathbf{s},\mathbf{t})}_{\mathbf{l},\mathbf{j}}$ is highest. Remember this is our suspect point. We can then simply gather all versions of the solution on that point, $u_{\mathbf{l}'}(x^*_{\mathbf{l},\mathbf{j}})$, with $\mathbf{l}' \geq \mathbf{l}$ and $\mathbf{l}', \mathbf{l} \in \mathcal{I}_g$. We then try to fit these function values to a constant $\tilde{u}$, since we know they should all be similar:

$$u_{\min} \leftarrow \min_{\tilde{u}} \sum_{\substack{\mathbf{l}' \geq \mathbf{l} \\ \mathbf{l}', \mathbf{l} \in \mathcal{I}_g}} \rho\left(u_{\mathbf{l}'}(x^*_{\mathbf{l},\mathbf{j}}) - \tilde{u}\right). \tag{6.36}$$

Finally, just as we did in the previous section, we can normalize the residuals

$$r_{\mathbf{i}} = u_{\mathbf{i}}(x^*_{\mathbf{l},\mathbf{j}}) - u_{\min}$$

according to Eq. (6.34), substituting $|\mathcal{V}|$ by the number of component solutions that contain grid point $x^*_{\mathbf{l},\mathbf{j}}$). The number of unknowns is not $d \cdot (n - \tau)$ anymore, but 1 (the value of the constant $\tilde{u}$).

The last remaining question is what to do when we do not have enough values of $u_{\mathbf{i}}(x^*_{\mathbf{l},\mathbf{j}})$ (at least five) to perform the robust fit. Our heuristic of checking for the decay of the surpluses didn't sound very robust. One possible solution – in fact, the most satisfactory we have tested – is to *switch* to our first sanity check in such cases. We will discuss how well this works in the results section.

Costs

This second algorithm has even lower costs than the first one. In both, we need to search for $x^*_{\mathbf{l},\mathbf{j}}$, which we argued is cheap and can be done in parallel. Then, within each group, we have to gather all the values of $u_{\mathbf{i}}(x^*_{\mathbf{l},\mathbf{j}})$, which are at most $|\mathcal{I}_g|$ and does not require communication across groups. The minimization problem (6.36) is as simple as it gets, fitting only a constant model. We therefore argue that the added costs of this algorithm are very low, which we will confirm in our experiments.

Parallelization

The search for $x^*_{\mathbf{l},\mathbf{j}}$ is performed in parallel just as with the first sanity check. The process that contains that grid point then solves the optimization problem (6.36).

6.3.4 Detection Rates

Some authors try to design SDC detection algorithms in such a way that a minimum detection rate can be guaranteed. For example, the authors in [GZP+16] define a maximum error they are willing to let undetected, and based on that they come up with a confidence interval for the error, assuming that it is normally distributed. Their technique requires an additional preprocessing step to learn how the error might behave. We have not yet investigated that possibility for our case (although it would be an interesting research topic), so it is not possible to estimate a detection rate of our algorithms a priori.

For the moment, we will instead inject SDC into our algorithms as extensively as possible and experimentally report on the detection rate we achieve.

6.3.5 Simulation Scenario: nD Advection-Diffusion Equation

Our initial implementations were tested on a linear 2D advection equation. Now that we have refined our algorithms and have described a way to parallelize them, we investigate an n-dimensional advection equation with an additional diffusion term. The PDE is given by

$$\begin{aligned} \partial_t u - \Delta u + \mathbf{a}\cdot\nabla u &= f \qquad \text{in } \Omega\times[0,T) \\ u(\cdot,t) &= 0 \qquad \text{in } \partial\Omega \end{aligned} \tag{6.37}$$

with $\Omega = [0,1]^d$, $t = [0, 0.05]$, $\mathbf{a} = \mathbf{1}^T$ and $u(\cdot,0) = e^{-100\sum_{i=1}^{d}(x_i-0.5)^2}$. The PDE was implemented using the framework `DUNE-pdelab`.Physically, Eq. (6.38) starts with a Gaussian function in the middle of the domain at time $t = 0$. The Gaussian then travels a constant velocity $\mathbf{a}$ in all d dimension. For the spatial discretization we used the FVE method on rectangular grids. For the integration in time we used an explicit Euler scheme.

We investigated the quality of our detection algorithms in 2, 3 and 5 dimensions. The five-dimensional case was also used to test the computational cost and scalability of our algorithms, since these are very expensive simulations. In every simulation scenario we performed 50 time steps with a step of $\Delta t = 10^{-3}$. The component solutions were combined every 10 time steps. The solution of the robust regression problems (6.31) and (6.36) were implemented using the GNU Scientific Library [GDT+15]. For the weight function w we used Cauchy function, and we set the value of the scaling constant to $C = 0.01$.

We simulated SDC in the same way as in our preliminary experiments: on each run, we randomly choose a component solution $u_{\mathbf{i}}$ which will be affected by SDC. Once again we restricted our attention to solutions with the highest resolution,

$|\mathbf{i}|_1 = n + d \cdot (\tau + 1) - 1$, since we have seen that it is in these solutions that SDC can be hardest to detect, which happens when the hierarchically highest subspaces are affected. We then choose a grid point $x_{\mathbf{l},\mathbf{j}}$ in $u_{\mathbf{i}}$ and alter its value in one of the three ways described in Section 6.2. The grid point is chosen to be either in the middle of the domain, $x_i = 0.5,\ \ i = 1, \ldots, d$ (corresponding to the hierarchically lowest subspace) or near the middle, at $x_i = 0.5 - h_i,\ \ i = 1, \ldots, d$ (corresponding to the hierarchically highest subspace).

For all these possible scenarios, we choose a time iteration where SDC occurs and we do this only once during the whole simulation. We then investigate whether SDC is detected before combining the component solutions and we look at the approximation error of the combination solution at the end of the simulation, comparing it with a reference solution without faults.

6.3.6 Results: Detection Rates and Error

2D Case

for the 2d case we chose the combination parameters $n = 5$ and $\tau = 2$, resulting in a combination technique with 14 component solutions. our results for the approximation error can be found in fig. 6.12. the top three subfigures correspond to the case where sdc is injected in the middle of the unit square and the bottom three to sdc injected near the middle of the domain. the three subplots correspond to the three different magnitudes of sdc (10^{-300}, $10^{-0.5}$ and 10^{+150}). the x-axis represents the iteration at which sdc was injected, and on the y-axis we plot the l_2 relative error $e = \frac{\|u_n^{(c)} - u_{\text{ref}}\|_2}{\|u_{\text{ref}}\|_2}$ at the end of the 50 iterations compared to a reference solution of level $\mathbf{n}' = (7, 7)$.

four different measurements can be observed in each plot. the solid blue line is the error of the combination technique compared to the full grid reference solution where no sdc occurs. red crosses indicate the error of the combination technique when sdc occurs and is not detected. green stars show the error of the combination when using the first sanity check to detect sdc, and the empty circles correspond to the second detection method being turned on.

let us start with the three top subfigures (sdc in the middle of the domain). we can see that, when sdc occurs, the error is roughly twice as large for moderate magnitudes of sdc (10^{-300} and $10^{-0.5}$). when sdc is large (10^{+150}) the error is also of that order of magnitude, so we did not plot it with the rest of the data. the first method detected the sdc in 88%, 64% and 100% of the cases, corresponding to the three magnitudes of sdc. method 2 had a detection rate of 98%, 84% and 100%. but more importantly, in the scenarios where sdc was not detected, the approximation error is still quite close to the fault-free case.

the three bottom figures show the error when injecting sdc near the middle of the domain. the detection rates were 22%, 10% and 100% for our first detection algorithm and 0%, 0% and 100% for the second. the detection rates are therefore

4

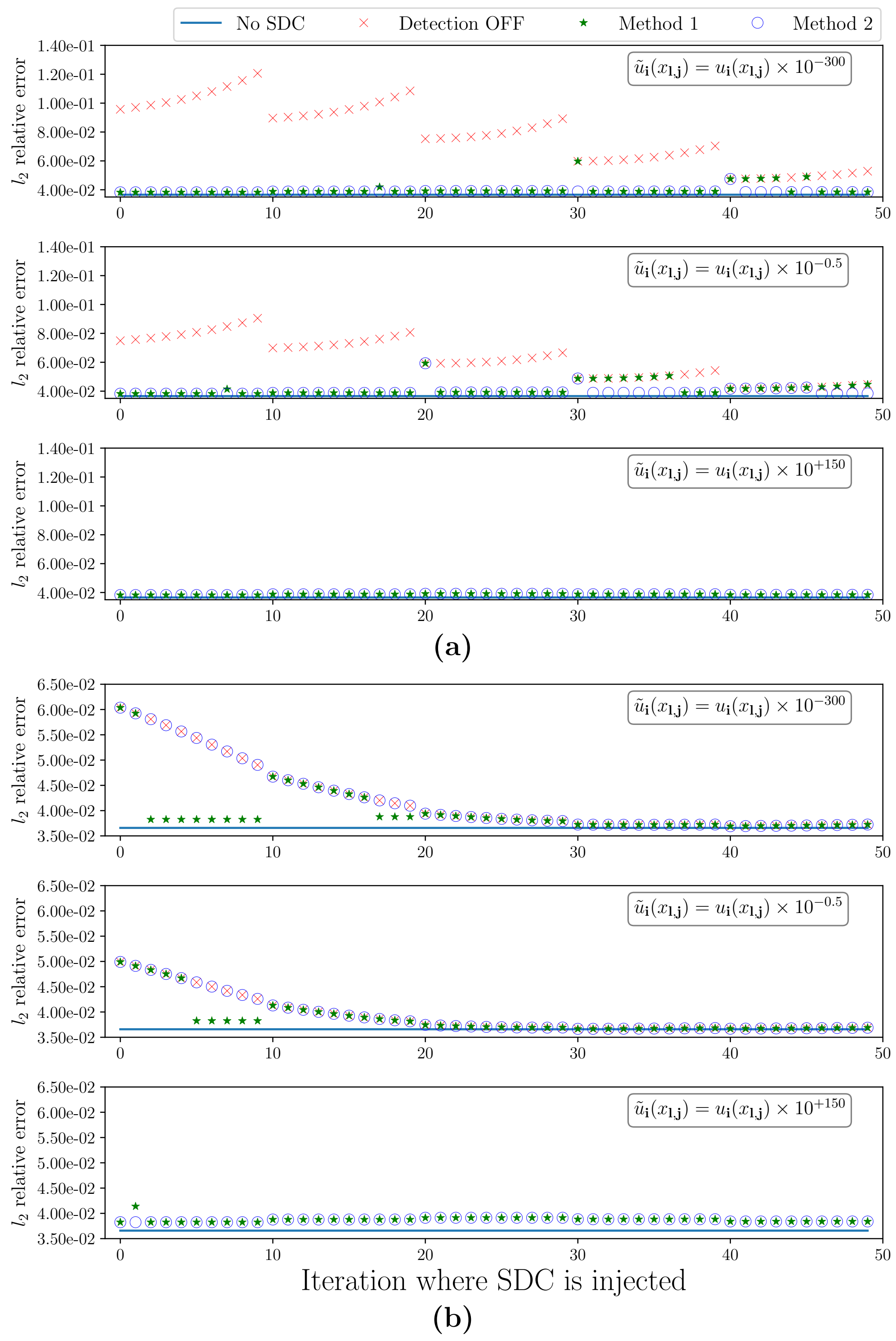


Figure 6.12: Approximation error of the combination technique in 2D ($n = 5$, $\tau = 2$) injecting SDC (a) in the middle and (b) near the middle of the square domain. (With data as submitted in [PHBP].)

quite lower, but once again it is important to notice that the error remains small, and that large sdc (10^{+150}) is always detected.

3d case

We increased the dimension to three and used the combination parameters $n = 3$ and $\tau = 2$, which gives 10 component solutions. Since these simulations are already considerably more expensive than in two dimensions, we injected SDC at only six iterations for every case (as opposed to doing it at every iteration), namely, at iterations 0, 9, 19, 29, 39 and 49. We calculated the error of the solution compared to a full grid solution of level $\mathbf{n}' = (5, 5, 5)$ at the end of each simulation. The results can be seen in Fig. 6.13, and similar conclusions can be drawn as with the 2D case, but in 3D, our first method performs worse for moderate SDC (10^{-300} or $10^{-0.5}$). Luckily, our second sanity check still worked very well, detecting all SDC that would have otherwise led to large errors.

5D Case

We finally increased the dimension to $d = 5$, keeping the combination parameters $n = 3$ and $\tau = 2$. This results in a combination with 21 component solutions. The reference solution has level $\mathbf{n}' = (5, 5, 5, 5, 5)$. We injected SDC as in the 3D case, at iterations 0, 9, 19, 29, 39 and 49, but we only considered one scenario, namely, injecting SDC in the middle of the domain and detecting using the second method. This is due to the fact that each run can take up to several hours. Figure. 6.14 shows our results. As in the previous cases, we were able to detect SDC in every occasion. For the parallelization we used two process groups, each one consisting of 1024 processes.

6.3.7 Results: Scaling

Earlier we argued that our detection algorithms should be cheap, at least compared to the computational resources required by other steps of the combination technique (especially solving the PDE on the component grids). In order to confirm this, we used a combination technique with high resolution ($n = 5$, $\boldsymbol{\tau} = (3, 3, 2, 2, 2)$), which approximates a full grid solution of level $\mathbf{n}' = (8, 8, 7, 7, 7)$ and is made up of 126 component grids.

To test how well the different parts of the algorithm scale, we increased the number of processes per group, using 256, 512, 1024, 2048 and 4096 processes in every of the 8 groups used, so the total number of processes ranges from 2048 to 32768.

We are primarily interested in the cost of detecting and recovering from SDC compared to the most expensive steps of the combination technique. In Fig. 6.14 we plot our time measurements for three different steps of the algorithm:

1. *Solve* tells us the time required by the DUNE framework to perform one time step of Eq. (6.38).

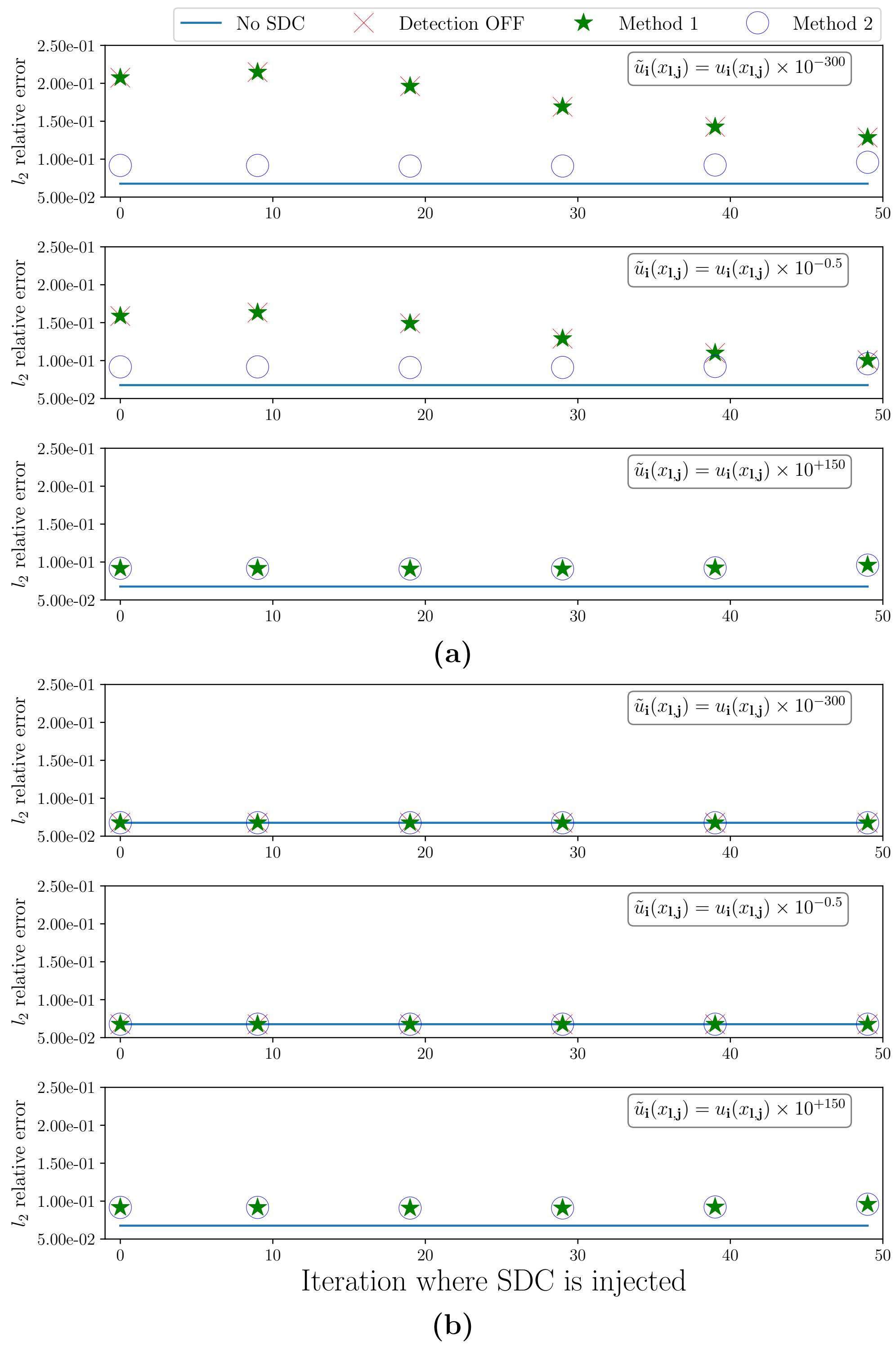


Figure 6.13: Approximation error of the combination technique in 3D ($n = 3$, $\tau = 2$) injecting SDC (a) in the middle and (b) near the middle of the square domain. (With data as submitted in [PHBP].)

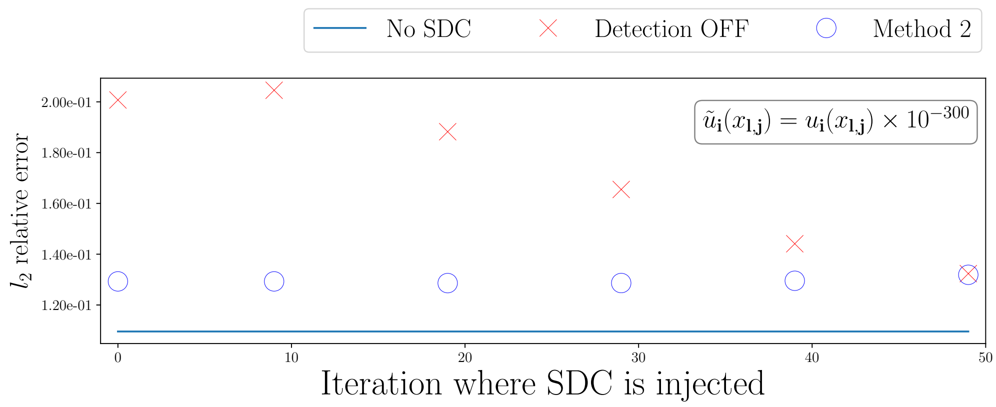


Figure 6.14: Approximation error of the combination technique in 5D ($n = 3$, $\tau = 2$) injecting SDC in the middle. (With data as submitted in [PHBP].)

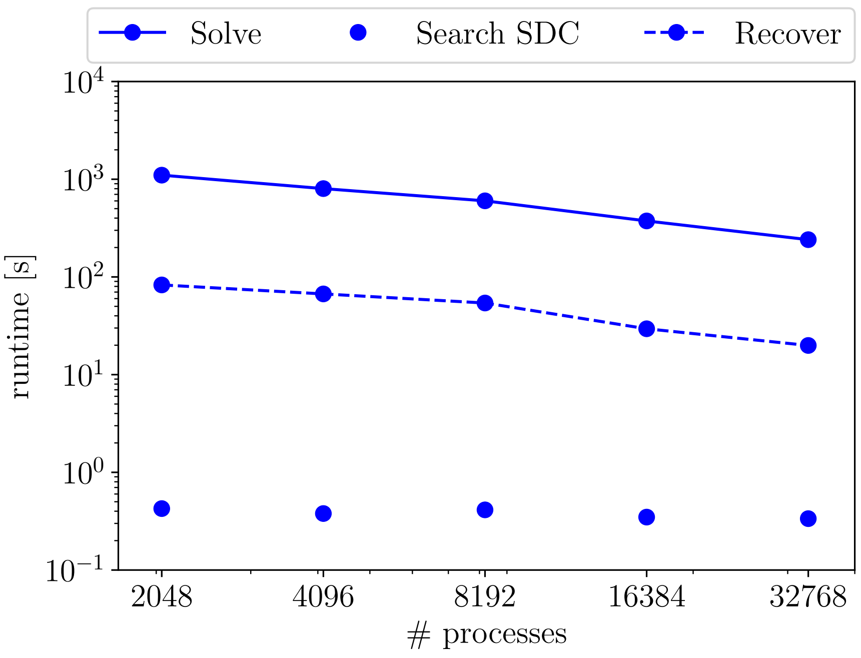


Figure 6.15: Scaling experiments with SDC detection on. We used 8 process groups and varied the number of processes per group, doubling from 256 until 4096. The total computational time required to ensure silent error resilience (*Search SDC* + *Recover*) is still one order of magnitude smaller than one single time step of the PDE solver (As submitted in [PHBP].)

2. *Recover* indicates the time to remove the solution with SDC, adapting the combination coefficients, and restarting the task with the correct values for the next combination step.

3. *Search SDC* is the time needed to carry out all steps of our second SDC detection method (see Section 6.3.3).

The results for the *Solve* and *Recover* steps closely resemble those of Chapter

$\mathbf{i}$	$u_{\mathbf{i}}(x^*_{\mathbf{i},\mathbf{j}})$
(3, 3, 3, 3, 3)	0.94203
(3, 3, 4, 3, 3)	0.94203
(4, 3, 3, 3, 3)	0.94203
(3, 3, 3, 3, 4)	**0.94571**
(3, 4, 3, 3, 4)	0.94203
(3, 3, 3, 4, 3)	0.94203

Table 6.2: Example of measurements that lead to false positives, here at point $x^*_{\mathbf{i},\mathbf{j}} = (0.5, 0.5, 0.5, 0.5, 0.5)$. Since the value corresponding to $u_{(3,3,3,3,4)}$ is different from all others, it can be marked as an outlier, even though its value is not wrong.

5, where we recovered from hard faults in parallel. The total time needed to detect and react to SDC is the sum of the *Search SDC* and *Recover* steps. The former is, as we can observe from the difference of 3 to 4 orders of magnitude, negligible (and decreases slightly with increasing processor count), whereas the latter is dominated by the time required to reinitialize the task identified with SDC, as was also the case with hard faults. One main difference with our algorithm to detect from hard faults is that wrong component solutions do not have to be redistributed to a new group, since all processes continue to work as usual.

We consider these results to be promising. It does require some additional effort to come up with a detection strategy specific to the algorithm under consideration (such as the combination technique), but we can see that it can pay off in terms of the computational overhead required in the end. This seems to confirm the position that algorithmic fault tolerance is arguably the most attractive alternative to resilience in future exascale systems.

6.3.8 Dealing with False Positives

There is interesting behavior of the combination technique that we have left out of the discussion until now, but which caused some problems for while detecting SDC in higher dimensions. When we increased the dimension to $d = 5$, we encountered a growing number of false positives while searching for SDC, that is, component solutions that were wrongly marked as having been affected by SDC. We observed this when the function value $u_{\mathbf{i}}(x^*_{\mathbf{i},\mathbf{j}})$ was almost identical across several component solutions but slightly different in others. An example of such a measurement can be found in Table 6.2.

Since we penalize outliers strongly (by choosing a small normalization constant C), even small variations such as this one are assigned a very large standardized residual, which results in the measurement being marked as outlier. The values cluster in this way probably because we combined after a small number of time steps, which may cause each component solution to vary only slightly in some regions from one combination step to the next.

We managed to address this problem as follows. If we start with the six measurements from Table 6.2 and we run the outlier detection algorithm, the fourth measurement is marked as an outlier. At this point we perform two measurements. First, we combine all the function values using their classical combination coefficients,

$$u_c = \sum_{\mathbf{i}} c_{\mathbf{i}} u_{\mathbf{i}}(x^*_{\mathbf{l},\mathbf{j}}).$$

Then we use the Fault Tolerant Combination Technique to find alternative combination coefficients to exclude the suspicious value and combine with the new combination coefficients,

$$u'_c = \sum_{\mathbf{i}} c'_{\mathbf{i}} u_{\mathbf{i}}(x^*_{\mathbf{l},\mathbf{j}}).$$

We can now compare u_c and u'_c. If they are similar, it means that the suspicious value is not really an outlier, whereas if they are very different, it might be. We can therefore compute their relative error,

$$e_{\text{rel}} = \frac{|u_c - u'_c|}{|u_{\min}|},$$

with $u_{\min}$ being the solution of the regression problem (6.36). If this error is small (for example, smaller than 5%), we can safely conclude that the value we're suspicious of is not in fact an outlier. Even if the difference was indeed caused by SDC, as long as the relative error is small, it is fine to ignore it and combine as usual, since we're making sure that the error introduces is small. We applied this check during all our simulation from the previous section, and it was particularly useful for the 5D case.

6.4 SDC Detection via Quantities of Interest

Our starting point to detect SDC in the combination technique was to look at the function values of the different component solutions, and trying to determine whether large differences are within acceptable bounds. But there might be cases where the function values themselves might not be of primary interest, but rather a given Quantity of Interest (QoI) Q related to the numerical solution $u_{\mathbf{i}}$, or $Q(u_{\mathbf{i}})$. The QoI could be many things, and it usually depends on the PDE being solved. For example, as we mentioned in Chapter 4, plasma physicists working with `GENE` are often interested in integrated quantities (or *observables*) such as the mean parallel velocity or the radial particle flux, all of which are scalars obtained by integrating the solution fields in different ways.

Two possible advantages of investigating QoIs instead of the solution fields themselves are the fact that QoIs are scalars, and not high-dimensional fields like $u_{\mathbf{i}}$, and that if SDC occurs but these quantities are correct, we could ignore it and not spend any time trying to search and correct it. Additionally, we would not have

$\mathbf{i}$	$Q_h(u_\mathbf{i})$	$\hat{r}_\mathbf{i}$
(7, 3)	3.14175	0.01204
(5, 4)	3.14268	1.32745
(4, 6)	3.14215	0.67156
(6, 4)	3.14215	0.67156
(4, 5)	3.14268	1.32745
(4, 3)	3.14245	0.23318
(4, 4)	3.14344	2.19886
(5, 5)	3.14222	0.79327
(6, 3)	**3.13748**	**7.66325**
(3, 6)	3.14189	0.00742
(3, 7)	3.14175	0.01204
(3, 5)	3.14215	0.01368
(3, 4)	3.14245	0.23318
(5, 3)	3.14215	0.01368

Table 6.3: Example of SDC detection using QoIs (in this case, integrating the solution over the domain). The standardized residuals reveal $u_{(6,3)}$ to be wrong.

to worry anymore about where exactly SDC occurs in the solution field.[5] However, maybe there are ways in which a solution field could be critically wrong, but in a way that is not reflected on the QoIs. The wrong data could then propagate further after deciding to ignore it, which could ruin the combination. We think such a scenario is unlikely, but it is important to keep it in mind.

The ideas of robust regression presented so far can be directly applied to the quantities of interest. We simple need a model for the error expansion of the QoI, $Q(u) - Q(u_\mathbf{i})$, or for the QoI itself (say, we expect it to be constant across all component solutions).

A simple example could be QoI defined as the integral of the field over the domain [Wak03]:

$$Q(u) = \int_\Omega u(\vec{x})\mathrm{d}\vec{x}. \tag{6.38}$$

We can then use the combination technique to approximate u (or $Q(u)$) and perform robust regression to search for outliers. Say we approximate (6.38) using a trapezoidal rule Q_h, for which we know that the error will be of order 2 in in the mesh size $h = 1/N$, with $N = \prod_{j=1}^{d}(2^{i_j} + 1)$ being the total number of points, so we have

$$Q(u) - Q_h(u_\mathbf{i}) = C \cdot h^2 + \mathcal{O}(h^3). \tag{6.39}$$

So for the robust minimization problem we could, for example, try to fit our measurements of $Q_h(u_\mathbf{i})$ to a polynomial model of powers of h,

$$Q_h(u_\mathbf{i}) \approx \tilde{Q}(\vec{c}, h) := Q(u) + \sum_{j=0}^{p} c_j h^j. \tag{6.40}$$

[5]In general, we think it would be interesting to investigate variants of the detection algorithms we have described which do not depend on where in the solution the fault occurred.

This would result in the following robust minimization problem:

$$\min_{\vec{c}} \sum_{\mathbf{i}\in\mathcal{I}} \rho\left(Q_h(u_{\mathbf{i}}) - \tilde{Q}(\vec{c}, h)\right). \tag{6.41}$$

We implemented this detection algorithm for the 2D linear advection equation described in Section 6.2, using a combination technique with $n = 5$ and $\tau = 2$. In one simulation scenario, we injected SDC of magnitude $10^{-0.5}$ into one component solution and computed the QoI (6.38) using a two-dimensional trapezoidal rule. The results of the numerical integral can be seen in Table 6.3. The initial condition was chosen such that the exact integral $Q(u)$ is equal to π. For the polynomial model (6.40) we chose $p = 2$, so the model has three unknowns c_0, c_1 and c_2. Solving the robust minimization problem and computing the standardized residuals as before reveals the outlier measurement, which corresponds to component solution $u_{(6,3)}$.

6.5 Evaluation and Comparison

In Section 2.3.1 we briefly discussed some techniques that have been used by others to deal with silent faults. A standard approach consists of performing checkpoints and adding a verification step to make the checkpoints are fault-free. This requires estimating the meant time between errors for a given system, which does not seem easy. In a sense, our approach is somewhat similar if we consider every combination solution as a checkpoint, and the sanity checks are the verification mechanism. One advantage of our approach is that we do not need to keep multiple checkpoints, since we verify at every combination step that the combination solution is not tainted.

Classical ABFT uses elaborate checksum protocols to make sure that all computations are correct. (see [SWA+14], Section 5.4.2 for a comprehensive list of examples.) We try to avoid such an approach because, as we argued in Section 2.3.2, the better question to ask is whether numerical errors are bounded, not whether every computation is performed exactly.

One final common approach is replication, which comes at a cost of sacrificing computing resources for the replicas (see [vDVDJ13]). In this section we have argued that the hierarchical structure of sparse grids (along with the error splitting assumption) can give us enough information regarding the correctness of the component solutions, so adding replication does not seem necessary. In a way, the fact that the FTCT adds some more component solutions to the combination technique is a way to ensure that we have enough information to combine properly in case of faults. Although this is different from replication, some analogies could be drawn. Some of the most promising approaches seem to be algorithm based, and we believe this is a good approach for the combination technique (see [DHR15], Section 6.2 for more examples).

7 Conclusions

In this work we adapted the combination technique – an extrapolation algorithm to solve high-dimensional problems – to allow it to tolerate different types of faults in HPC systems. After arguing in favor of algorithm-based techniques to develop resilient codes, we discussed the numerical properties of the combination technique that can be exploited to make it tolerate hard faults.

Preliminary experiments with the plasma simulation code `GENE` showed that the fault-tolerant combination technique had a low computational overhead and that its approximation quality was very close to the solution without faults. We then described how to adapt a massively-parallel implementation of the combination technique to make it fault tolerant. In our parallel experiments with `DUNE`, we confirmed the results from our preliminary experiments (low overhead and good approximation quality), but we also showed that the algorithms as we implemented them scale well. We are now at a point where large-scale simulations can be run reliably using the parallel framework, and ongoing experiments with `GENE` aim to verify this for realistic plasma simulations.

After discussing the problem of silent errors and some best practices to simulate and overcome them, we dived into the numerics of the combination technique and applied well-known techniques from robust regression to try to detect this type of errors. Our preliminary experiments with a 2D linear advection equation showed good detection rates but at a high computational cost. We then refined our algorithms and parallelized them in order to be able to test them on the parallel framework. Our results in up to 5D indicate that the detection rates remained high and that the computational overhead low. Our algorithms are still to be tested on more realistic simulation scenarios such as `GENE`. This is an interesting avenue for future research.

It is important to keep in mind the large uncertainty surrounding the type faulty behavior one can expect future exascale machines to exhibit. As mentioned in Chapter 2, software errors account for a considerable percentage of failures in a number of systems, and our experience working with the *Hazel Hen* supercomputer throughout this thesis confirms this. It might still be the case that node failures or silent data corruption become commonplace at exascale, but until then, it is pertinent not to extrapolate current trends too far into the future.

Equally important is the fact that exascale machines will call for exascale applications. Even nowadays, not many applications can make use of all resources available in petascale systems, and even fewer can use those resources efficiently. Some of the issues described in this thesis are expected to arise not simply from increased node count and system complexity: we have assumed that future applications will actually use all those resources, and possibly for long times. Otherwise, there would not be much difference between running simulations on exascale machines or on existing petascale computers.

These words of caution notwithstanding, the consensus of the HPC community regarding fault tolerance is a strong indicator that these problems should be taken seriously. And even if some claims end up being proven false with the arrival of exascale computers, fault tolerance has already opened new interesting avenues of interdisciplinary research that can prove valuable in other areas of HPC.

Bibliography

[AGJT15] Jeremie Abiteboul, Tobias Görler, Frank Jenko, and Daniel Told. Global electromagnetic simulations of the outer core of an asdex upgrade l-mode plasma. *Physics of Plasmas*, 22(9):092314, 2015.

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[B$^+$08] Peter Bastian et al. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2-3):103–119, 2008.

[B$^+$12] Wesley Bland et al. A proposal for user-level failure mitigation in the mpi-3 standard. *University of Tennessee*, 2012.

[BBD$^+$08] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Ralf Kornhuber, Markus Ohlberger, and Oliver Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82(2-3):121–138, 2008.

[BBD$^+$16] 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.

[BBGD$^+$15] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 275–278, New York, NY, USA, 2015. ACM.

[BBH$^+$13] Wesley. Bland, Aurélien. Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 2013.

[BBH$^+$14] George Bosilca, Aurélien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. Assessing the impact of ABFT and checkpoint composite strategies. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 679–688. IEEE, 2014.

[BCL95] Mary Ann Branch, Thomas F Coleman, and Yuying Li. A subspace, interior, and conjugate gradient method for large-scalebound-constrained minimization problems. Technical report, Cornell University, 1995.

[BDDL09] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 6(4):410 – 416, 2009.

[BFHH12] Patrick G Bridges, Kurt B Ferreira, Michael A Heroux, and Mark Hoemmen. Fault-tolerant linear solvers via selective reliability. *arXiv preprint arXiv:1206.1390*, 2012.

[BG04] Hans-Joachim. Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[BHM10] Peter Bastian, Felix Heimann, and Sven Marnach. Generic implementation of finite element methods in the distributed and unified numerics environment (DUNE). *Kybernetika*, 46(2):294–315, 2010.

[BSS15] Austin R Benson, Sven Schmit, and Robert Schreiber. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, 29(4):403–421, 2015.

[BY01] Yoav Benjamini and Daniel Yekutieli. The control of the false discovery rate in multiple testing under dependency. *Annals of statistics*, pages 1165–1188, 2001.

[C$^+$09] Franck Cappello et al. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 2009.

[C$^+$14] F. Cappello et al. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[CD05] Lu Charng-Da. *Scalable diskless checkpointing for large parallel systems.* PhD thesis, 2005.

[Che11] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 73–84. ACM, 2011.

[Che13] Zizhong Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices*, volume 48, pages 167–176. ACM, 2013.

[CPHM08] Cristian Constantinescu, Ishwar Parulkar, Rick Harper, and Sarah Michalak. Silent data corruptionmyth or reality? In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 108–109. IEEE, 2008.

[CSM14] Joseph Callenes-Sloan and Hugh McNamara. Algorithm selection for error resilience in scientific computing. In *Dependable Computing (PRDC)*, pages 96–105. IEEE, 2014.

[DB$^+$11] J. Dongarra, P. H. Beckman, et al. The international exascale software project roadmap. *IJHPCA*, 25(1):3–60, 2011.

[DBM$^+$11] 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. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[DHR15] Jack Dongarra, Thomas Herault, and Yves Robert. Fault tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[DM$^+$14] Catello Di Martino et al. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Dependable Systems and Networks (DSN)*, pages 610–621. IEEE, 2014.

[EBEG$^+$08] EN Elnozahy, Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godfrey, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, JS Plank, Partha Ranganathan, et al. System resilience at extreme scale. *Defense Advanced Research Project Agency (DARPA), Tech. Rep*, 2008.

[EHM14a] James Elliott, Mark Hoemmen, and Frank Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1193–1202. IEEE, 2014.

[EHM14b] James Elliott, Mark Hoemmen, and Frank Mueller. Resilience in numerical methods: A position on fault models and methodologies. *arXiv preprint arXiv:1401.3013*, 2014.

[FME$^+$12] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 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*, page 78. IEEE Computer Society Press, 2012.

[Gar07] Jochen Garcke. A dimension adaptive sparse grid combination technique for machine learning. *ANZIAM Journal*, 48:725–740, 2007.

[Gar13] Jochen Garcke. Sparse grids in a nutshell. In *Sparse grids and applications*, pages 57–80. Springer, 2013.

[GDT$^+$15] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth, and Fabrice Rossi. GNU scientific library reference manual. *Library available online at http://www. gnu. org/software/gsl*, 2015.

[Gei11] Al Geist. What is the monster in the closet? In *Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking*, volume 2, 2011.

[GG00] Jochen Garcke and Michael Griebel. On the computation of the eigenproblems of hydrogen and helium in strong magnetic and electric fields with the sparse grid combination technique. *Journal of Computational Physics*, 165(2):694–716, 2000.

[GGT01] Jochen Garcke, Michael Griebel, and Michael Thess. Data mining with sparse grids. *Computing*, 67(3):225–253, 2001.

[GH07] Michael Griebel and Jan Hamaekers. Sparse grids for the schrödinger equation. *ESAIM: Mathematical Modelling and Numerical Analysis*, 41(2):215–247, 2007.

[GHZ96] M. Griebel, Walter Huber, and C. Zenger. Numerical turbulence simulation on a parallel computer using the combination method. In *Flow simulation on high performance computers II*, pages 34–47, 1996.

[GLB$^+$11] Tobias Goerler, Xavier Lapillonne, Stephan Brunner, Tilman Dannert, Frank Jenko, Florian Merz, and Daniel Told. The global version of the gyrokinetic turbulence code GENE. *J. Comput. Phys.*, 230:7053–7071, 2011.

[glp] GNU linear programming kit, version 4.35. `http://www.gnu.org/software/glpk/glpk.html`.

[Gri92] Michael Griebel. The combination technique for the sparse grid solution of PDEs on multiprocessor machines. In *Parallel Processing Letters*, pages 61–70, 1992.

[GSZ92] Michael Griebel, Michael Schneider, and Christoph Zenger. A combination technique for the solution of sparse grid problems. In *Iterative Methods in Linear Algebra*, pages 263–281, 1992.

[GZP$^+$16] Pierre-Louis Guhur, Hong Zhang, Tom Peterka, Emil Constantinescu, and Franck Cappello. Lightweight and accurate silent data corruption detection in ordinary differential equation solvers. In *European Conference on Parallel Processing*, pages 644–656. Springer, 2016.

[H$^+$15] Brendan Harding et al. Fault tolerant computation with the sparse grid combination technique. *SIAM Journal on Scient. Comp.*, 37(3):C331–C353, 2015.

[HA84] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[Har15] Brendan Harding. Adaptive sparse grids and extrapolation techniques. In *Sparse Grids and Applications*. Springer, 2015.

[Har16a] Brendan Harding. Combination technique coefficients via error splittings. *ANZIAM Journal*, 56:355–368, 2016.

[Har16b] Brendan et al. Harding. Fault tolerant computation of hyperbolic partial differential equations with the sparse grid combination technique. 2016.

[Hee17] Mario Heene. *A massively parallel combination technique for the solution of high-dimensional PDEs.* Phd thesis, Universität Stuttgart, 2017.

[Heg03] Markus Hegland. Adaptive sparse grids. *ANZIAM Journal*, 44:C335–C353, July 2003.

[HGC07a] Markus Hegland, Jochen Garcke, and Vivien Challis. The combination technique and some generalisations. *Linear Algebra Appl.*, 420(2–3):249–275, 2007.

[HGC07b] Markus Hegland, Jochen Garcke, and Vivien Challis. The combination technique and some generalisations. *Linear Algebra and its Applications*, 420(2-3):249–275, 2007.

[HH13] Brendan Harding and Markus Hegland. A robust combination technique. *ANZIAM Journal*, 54:C394–C411, 2013.

[HHJP16] Philipp Hupp, Mario Heene, Riko Jacob, and Dirk Pflüger. Global communication schemes for the numerical solution of high-dimensional PDEs. *Parallel Computing*, 52, 2016.

[HJ] Philipp Hupp and Riko Jacob. Cache-optimal component grid hierarchization outperforming the unidirectional algorithm. submitted to the 22nd European Symposium on Algorithms (ESA 2014).

[HKP13] Mario Heene, Christoph Kowitz, and Dirk Pflüger. Load balancing for massively parallel computations with the sparse grid combination technique. In *Parallel Computing: Accelerating Comp. Science and Eng.*, pages 574–583, 2013.

[HP16a] 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*, 2016.

[HP16b] Mario Heene and Dirk Pflüger. Scalable algorithms for the solution of higher-dimensional PDEs. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 165–186. Springer, 2016.

[HPHBP16] Mario Heene, Alfredo Parra Hinojosa, Hans-Joachim Bungartz, and Dirk Pflüger. A massively-parallel, fault-tolerant solver for high-dimensional pdes. In *European Conference on Parallel Processing*, pages 635–647. Springer, 2016.

[Hup13] Philipp Hupp. Hierarchization for the sparse grid combination technique. *CoRR abs/1309.0392*, 2013.

[Hup14] Philipp Hupp. Performance of unidirectional hierarchization for component grids virtually maximized. In *ICCS 2014*, Procedia Computer Science. Elsevier, June 2014.

[HW77] Paul W Holland and Roy E Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-theory and Methods*, 6(9):813–827, 1977.

[ite] ITER. `www.iter.org`. Accessed: 2017-04-25.

[J$^+$00] Frank Jenko et al. Electron temperature gradient driven turbulence. *Physics of Plasmas (1994-present)*, 7(5):1904–1910, 2000.

[KGHW15] Egemen Kolemen, David A Gates, David A Humphreys, and Michael L Walker. Plasma control for iter and future fusion power plants. In *Plasma Sciences (ICOPS), 2015 IEEE International Conference on*, pages 1–1. IEEE, 2015.

[KH13] Christoph Kowitz and Markus Hegland. The sparse grid combination technique for computing eigenvalues in linear gyrokinetics. *Procedia Computer Science*, 18(0):449 – 458, 2013.

[KH14] Christoph Kowitz and Markus Hegland. An Opticom Method for Computing Eigenpairs. In Jochen Garcke and Dirk Pflüger, editors, *Sparse Grids and Applications - Munich 2012 SE*, volume 97 of *Lecture Notes in Computational Science and Engineering*, pages 239–253. Springer International Publishing, 2014.

[Kow16] Christoph Kowitz. *Applying the sparse grid combination technique in Linear Gyrokinetics*. Dissertation, 2016.

[KPJH12] Christoph Kowitz, Dirk Pflüger, Frank Jenko, and Markus Hegland. The combination technique for the initial value problem in linear gyrokinetics. In *Sparse Grids and Applications*, volume 88 of *Lecture Notes in Computational Science and Engineering*, pages 205–222, Heidelberg, October 2012. Springer.

[Mer09] Florian Merz. *Gyrokinetic simulation of multimode plasma turbulence*. PhD thesis, 2009.

[MF10] Bernd Mohr and Wolfgang Frings. Jülich Blue Gene/P extreme scaling workshop 2009. Technical report, Technical report FZJ-JSC-IB-2010-02. Online at http://juser.fz-juelich.de/record/8924/files/ib-2010-02.ps.gz, 2010.

[NTJ$^+$16] Alejandro Bañón Navarro, Daniel Told, Frank Jenko, Tobias Görler, and Tim Happel. Comparisons between global and local gyrokinetic simulations of an asdex upgrade h-mode plasma. *Physics of Plasmas*, 23(4):042312, 2016.

[OPHH$^+$] Michael Obersteiner, Alfredo Parra Hinojosa, Mario Heene, Hans-Joachim Bungartz, and Dirk Pflüger. A highly-scalable, algorithm-based fault-tolerant solver for gyrokinetic plasma simulations. Submitted to the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA'17).

[Pfl10] Dirk Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, München, August 2010.

[PHBP] Alfredo Parra Hinojosa, Hans-Joachim Bungartz, and Dirk Pflüger. Scalable algorithmic detection of silent data corruption for high-dimensional PDEs. In *Sparse Grids and Applications Workshop (SGA'16)*. Accepted.

[PHHHB16] Alfredo Parra Hinojosa, Brendan Harding, Markus Hegland, and Hans-Joachim Bungartz. Handling silent data corruption with the sparse grid combination technique. In *Proceedings of the SPPEXA Workshop*, LNCSE. Springer-Verlag, 2016.

[PHKH$^+$15] Alfredo Parra Hinojosa, Christoph Kowitz, Mario Heene, Dirk Pflüger, and Hans-Joachim Bungartz. Towards a fault-tolerant, scalable implementation of GENE. In *Proceedings of ICCE 2014*, LNCSE. Springer-Verlag, 2015.

[Rei12] Christoph Reisinger. Analysis of linear difference schemes in the sparse grid combination technique. *IMA Journal of Numerical Analysis*, 2012.

[RL05] Peter J Rousseeuw and Annick M Leroy. *Robust regression and outlier detection*, volume 589. John Wiley & Sons, 2005.

[Rob16] Yves Robert. An overview of fault-tolerant techniques for HPC, 2016. `http://graal.ens-lyon.fr/~yrobert/europar16.pdf`.

[RW07] Christoph Reisinger and Gabriel Wittum. Efficient hierarchical approximation of high-dimensional option pricing problems. *SIAM Journal on Scientific Computing*, 29(1):440–458, 2007.

[SAH15] P. E. Strazdins, M. M. Ali, and B. Harding. Highly scalable algorithms for the sparse grid combination technique. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 941–950, May 2015.

[sci] Robust nonlinear regression in scipy. `http://scipy-cookbook.readthedocs.io/items/robust_regression.html`. Accessed: 2017-05-30.

[Smo63a] Sergey Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Mathematics, Doklady*, 4:240–243, 1963.

[Smo63b] Sergey Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. In *Dokl. Akad. Nauk SSSR*, volume 4, page 123, 1963.

[SOR$^+$03] Ramendra K Sahoo, Adam J Oliner, Irina Rish, Manish Gupta, José E Moreira, Sheng Ma, Ricardo Vilalta, and Anand Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–435. ACM, 2003.

[SP10] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *Proceedings of the 9th Python in Science Conference*, pages 57–61, 2010.

[SW10] Songqing Shan and G Gary Wang. Survey of modeling and optimization strategies to solve high-dimensional design problems with computationally-expensive black-box functions. *Structural and Multidisciplinary Optimization*, 41(2):219–241, 2010.

[SWA$^+$14] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28:129–173, 2014.

[vDVDJ13] Hubertus JJ van Dam, Abhinav Vishnu, and Wibe A De Jong. A case for soft error detection and correction in computational chemistry. *Journal of Chemical Theory and Computation*, 9(9):3995–4005, 2013.

[Wak03] MA Wakefield. *Bounds on quantities of physical interest.* PhD thesis, University of Reading, 2003.

[Wal16] Johannes Walter. Design and implementation of a fault simulation layer for the combination technique on hpc systems. Master's thesis, University of Stuttgart, 2016.

[Win11] Hugo Winter. Numerical advection schemes in two dimensions. `www.lancs.ac.uk/~winterh/advectionCS.pdf`, 2011.

[Zen91] Christoph Zenger. Sparse grids. In *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 241–251. Vieweg, 1991.

[ZLG$^+$10] Ziming Zheng, Zhiling Lan, Rinku Gupta, Susan Coghlan, and Peter Beckman. A practical failure prediction with location and lead time for blue gene/p. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 15–22. IEEE, 2010.

[ZNK12] 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.