A Highly Scalable, Algorithm-Based Fault-Tolerant Solver for Gyrokinetic Plasma Simulations

Michael Obersteiner
Technical University of Munich, Department of Informatics
Garching
oberstei@in.tum.de

Alfredo Parra Hinojosa
Technical University of Munich, Department of Informatics
Garching
hinojosa@in.tum.de

Mario Heene
University of Stuttgart, IPVS
Stuttgart
mario.heene@ipvs.uni-stuttgart.de

Hans-Joachim Bungartz
Technical University of Munich, Department of Informatics
Garching
bungartz@in.tum.de

Dirk Pflüger
University of Stuttgart, IPVS
Stuttgart
dirk.pflueger@ipvs.uni-stuttgart.de

ABSTRACT

With future exascale computers expected to have millions of compute units distributed among thousands of nodes, system faults are predicted to become more frequent. Fault tolerance will thus play a key role in HPC at this scale. In this paper we focus on solving the 5-dimensional *gyrokinetic Vlasov-Maxwell equations* using the application code GENE as it represents a high-dimensional and resource-intensive problem which is a natural candidate for exascale computing. We discuss the *Fault-Tolerant Combination Technique*, a resilient version of the Combination Technique, a method to increase the discretization resolution of existing PDE solvers. For the first time, we present an efficient, scalable and fault-tolerant implementation of this algorithm for plasma physics simulations based on a manager-worker model and test it under very realistic and pessimistic environments with simulated faults. We show that the *Fault-Tolerant Combination Technique* – an algorithm-based forward recovery method – can tolerate a large number of faults with a low overhead and at an acceptable loss in accuracy. Our parallel experiments with up to 32k cores show good scalability at a relative parallel efficiency of 93.61%. We conclude that algorithm-based solutions to fault tolerance are attractive for this type of problems.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**;
• **Applied computing** → *Physics*;

KEYWORDS

exascale, plasma simulations, fault tolerance, resilience, sparse grids, combination technique

ScalA17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5125-6/17/11.
https://doi.org/10.1145/3148226.3148229

ACM Reference Format:
Michael Obersteiner, Alfredo Parra Hinojosa, Mario Heene, Hans-Joachim Bungartz, and Dirk Pflüger. 2017. A Highly Scalable, Algorithm-Based Fault-Tolerant Solver for Gyrokinetic Plasma Simulations. In *Proceedings of ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3148226.3148229

1 INTRODUCTION

Experts widely agree that supercomputers will exhibit faulty behavior multiple times per day in the exascale era [5]. Current petascale systems already face this problem. The Tsubame 2 supercomputer reported 962 faults in 18 months, which means it had a mean time between failures (MTBF) of about 13.5 hours; the Blue Waters system suffered 2-3 node failures per day; and Titan supercomputer has also been reported to fail several times per day [20]. In exascale supercomputers with several million processors, jobs could fail as often as every 30 minutes [22]. For this reason, the fault tolerance community has been working hard to find efficient methods to deal with system errors. Examples are dedicated checkpointing protocols, fault prediction and replicating the data or hardware. We are particularly interested in *algorithm-based fault tolerance* (ABFT) to handle fail-stop errors when solving PDEs, thus exploiting the numerical properties of a specific algorithm to make it resilient to faults. In this work we consider only hard faults, e.g. process failures, but our framework is also capable of detecting and recovering from soft faults such as bit flips. The procedure to recover from soft faults is mostly identical to the case of hard faults but adds a very cheap routine that searches for data corruption within the grids (see [14]).

We focus on high-dimensional partial differential equations (PDEs) because they pose many challenges for HPC, the most obvious being the exponential increase in discretization points with the problem's dimensionality. For more than 3 dimensions, one quickly runs into memory problems. Such PDEs are commonplace in physics and require days to solve on full machines. These applications are, therefore, natural candidates for testing new fault tolerance techniques.

In this paper we consider one of these applications – the simulation of hot, magnetized plasma, whose study plays a central

role in developing plasma fusion reactors. We briefly introduce a well-known numerical method – the *Sparse Grid Combination Technique* – which can be used not only to increase the resolution of parallel PDE solvers, but also to make them fault-tolerant. We have argued in the past that the Combination Technique is a promising algorithm for future exascale systems. We have shown that the Combination Technique converges in the presence of faults affecting plasma simulations [18]. We have also implemented a highly-scalable version of the Combination Technique, which is application-independent [13]. Finally, we have put our parallel algorithms to test with a simple PDE solver (a linear advection-diffusion equation), showing that it both converges and scales [12]. In this paper we present a major milestone of our work: we apply the parallel *Fault-Tolerant Combination Technique* (FTCT) to large-scale plasma physics simulations, and test it in realistic (and pessimistic) fault scenarios. We argue that this implementation could be used in future exascale systems.

The novel contributions of this paper are as follows:

- Applying the FTCT to plasma physics in our highly scalable framework using dynamic communicator regrouping.
- Statistical error analysis of the FTCT for moderate to massively faulty environments for a realistic application.
- Comparison of the FTCT to the classical Combination Technique regarding scalability.

2 PLASMA SIMULATIONS WITH THE COMBINATION TECHNIQUE

Scientists building the plasma fusion reactor *ITER* describe the project as "a major step that may provide a future energy alternative for all humankind"[1]. With a projected cost of 14 billion USD shared among 35 countries, the prospect of clean, safe, and abundant energy obtained from plasma fusion seems closer than ever. Numerical simulations have played a key role in this field, since they are used to study both turbulent plasma flow and optimal reactor geometries. The former is currently a major challenge for plasma physicists. With temperatures reaching 100 million Kelvin, various anomalous transport phenomena arise that complicate the extraction of energy from plasma reactions. These turbulent processes can be better understood with the help of high-resolution simulations.

One of the most robust numerical codes dedicated to studying the physics of plasma turbulence is GENE [15]. The code solves the *gyrokinetic Vlasov-Maxwell equations*, a system of nonlinear integrodifferential equations of the general form

$$\frac{\partial u}{\partial t} = \mathcal{L}(u) + \mathcal{N}(u)\,, \tag{1}$$

where $\mathcal{L}$ and $\mathcal{N}$ are the linear and nonlinear parts of the differential operator, and $u \equiv u(x, y, z, v_\parallel, \mu; t)$ is the (5+1)-dimensional distribution function of the plasma field. GENE uses a 5D Cartesian grid to discretize the three spatial dimensions x, y, z and the two velocity dimensions $v_\parallel$ and μ. It also simulates the interaction of different *species* (such as ions or electrons), each of which has its own 5D grid. The domain – normally a part of a tokamak or stellarator fusion reactor – is parallelized in all dimensions using domain decomposition, and the time evolution is solved with a Runge-Kutta scheme of fourth order. Experiments with up to 262k cores have shown that GENE scales very well [17]. But despite being highly optimized, GENE can only resolve small cross-sections of the physical domains of large reactors (such as *ITER*), since every increase in the resolution of the 5D grid comes at a huge computational cost.

One way to address this problem is using extrapolation algorithms. This is the class of algorithms we are interested in. In particular, one can apply the *Sparse Grid Combination Technique* [8] to high-dimensional PDE solvers.

The idea is the following. Consider the exact solution of a PDE, u, and its numerical approximation $u_\mathbf{n}$ defined on a d-dimensional Cartesian grid $\Omega_\mathbf{n}$ with $(2^{n_1} \pm 1) \times \cdots \times (2^{n_d} \pm 1)$ grid points (± 1 depending on whether the grid has boundary points in a given dimension). We use boldface letters to denote d-dimensional multi-indices, e. g., $\mathbf{n} = (n_1, \ldots, n_d)$. The Combination Technique defines a *set of grids* $\Omega_\mathbf{i}$ with different discretization resolutions, all of them coarser (i. e., with fewer discretization points) than $\Omega_\mathbf{n}$. One then solves the PDE *on all grids* $\Omega_\mathbf{i}$, giving a set of solutions $u_\mathbf{i}$ which are *combined* together with certain weights $c_\mathbf{i}$,

$$u_\mathbf{n}^{(c)} = \sum_{\mathbf{i} \in \mathcal{I}} c_\mathbf{i} u_\mathbf{i}. \tag{2}$$

Here, $\mathcal{I}$ is the set of multi-indices that define the grids $\Omega_\mathbf{i}$, and $u_\mathbf{n}^{(c)}$ is the resulting *combination solution*, which should be a good approximation of the reference (full grid) solution $u_\mathbf{n}$,

$$u_\mathbf{n}^{(c)} \approx u_\mathbf{n}.$$

Given certain smoothness conditions, it is expected that the approximation quality is only slightly reduced, and that computing the individual $u_\mathbf{i}$ and combining them is cheaper than computing $u_\mathbf{n}$ directly. It might even be the case that computing $u_\mathbf{n}$ directly is not even possible, which is often the case for high-dimensional problems. We call each $u_\mathbf{i}$ a *component solution*, each $\Omega_\mathbf{i}$ a *component grid*, and each $c_\mathbf{i}$ a *combination coefficient*.

Evidently, only certain choices of the combination coefficients $c_\mathbf{i}$ result in a good extrapolation. The *Truncated Combination Technique* is one such choice, which is defined as

$$u_{n,\boldsymbol{\tau}}^{(c)} = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\mathbf{i} \in \mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}} u_\mathbf{i}, \tag{3}$$

where the index set is defined by

$$\mathcal{I}_{q,\boldsymbol{\tau}}^{d,n} = \{\mathbf{i} : |\mathbf{i}|_1 = n + (d-1) + |\boldsymbol{\tau}|_1 - q, \quad \mathbf{i} > \boldsymbol{\tau}\}, \tag{4}$$

and $\binom{d-1}{q}$ is a binomial coefficient. With this choice, $u_{n,\boldsymbol{\tau}}^{(c)}$ should approximate a full grid solution $u_{\mathbf{n}'}$ with $\mathbf{n}' = n \cdot \mathbf{1} + \boldsymbol{\tau}$, and the truncation parameter $\boldsymbol{\tau}$ defines a minimum resolution in each dimension. Comparisons with multi-indices are done component-wise, so $\mathbf{i} > \boldsymbol{\tau}$ means $i_k > \tau_k$ for all $k = 1, \ldots, d$.

As an example in two dimensions, consider computing the solution $u_{(4,4)}$. One could try to approximate it using the Combination Technique, for example, with the choice $n = 4$ and $\boldsymbol{\tau} = (0, 0)$. This

[1] https://www.iter.org/faq#Is_there_consensus_in_the_scientific_community_about_the_ITER_Project

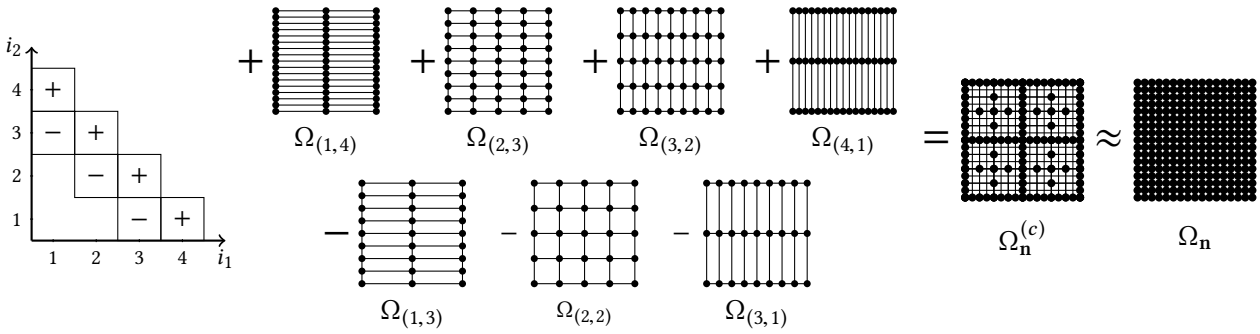

Figure 1: Grids resulting from a 2D Combination Technique with $n = 4$ and $\tau = (0, 0)$. The result of the combination is a sparse grid $\Omega_{\mathbf{n}}^{(c)}$.

would result in the following combination:

$$u^{(c)}_{4,(0,0)} = u_{(1,4)} + u_{(2,3)} + u_{(3,2)} + u_{(4,1)} - u_{(1,3)} - u_{(2,2)} - u_{(3,1)}.$$

This combination scheme is illustrated in Figure 1. Alternatively, one could increase the truncation parameter and choose $n = 3$, $\tau = (1, 1)$, resulting in the combination

$$u^{(c)}_{3,(1,1)} = u_{(2,4)} + u_{(3,3)} + u_{(4,2)} - u_{(2,3)} - u_{(3,2)}.$$

This combination uses fewer component solutions but they have higher resolution. In order to combine functions defined on grids with different resolutions, one can interpolate them all to the full grid resolution or combine them on the smallest function space containing all grid points of the component grids, which is a *sparse grid* [4]. This is where the method gets its name from.

In short, the Combination Technique is a way to approximate a (very expensive) full grid solution by solving the same PDE on a set of (cheap) anisotropic grids of lower resolution instead. Under certain smoothness assumptions, one can show that the approximation error of the Combination Technique solution $u^{(c)}_{n,\boldsymbol{\tau}}$ is in $O(h_n^2 \cdot (\log h_n^{-1})^{d-1})$, only slightly worse than that of the full grid solution $u_{\mathbf{n}}$, which is in $O(h_n^2)$ [8]. The Combination Technique has been used to solve a wide range of PDEs, from the Schrödinger equation [6] to turbulence problems with the Navier-Stokes equations [7].

Another advantage of using the Combination Technique to solve PDEs is that it offers an additional level of parallelism (in addition to any parallelization used to solve the individual component solutions $u_{\mathbf{i}}$). Since we solve the same PDE on each grid $\Omega_{\mathbf{i}}$, the $u_{\mathbf{i}}$ can be computed independently of each other, i. e., in parallel. Only their combination according to the formula (2) requires communication. This additional parallelism is why the Combination Technique is a good candidate to compute on future exascale computers [19].

3 PARALLEL FRAMEWORK AND FAULT TOLERANCE

3.1 Parallel Implementation of the Combination Technique

The Combination Technique can be parallelized in different ways (see [2] and [10, Chapter 3]). Our implementation is based on the manager-worker principle described in [13]. The main steps are shown in Algorithm 1. All but one processes are workers, and they are divided into G groups or process groups. Each group is responsible for calculating one or multiple component solutions $u_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{I}$. In other words, each process group is assigned a subset $\mathcal{I}_g$ of the whole set of indices $\mathcal{I} = \bigcup_{g=1}^{G} \mathcal{I}_g$, and their task is to compute all $u_{\mathbf{i}}$ in that subset. The remaining process is the manager. At initialization, it decides how to divide the workload among the groups (line 2). An efficient load balancing algorithm is described in [11]. The initial conditions are set for each solution (line 3), and the workers start solving their subset of solutions, one after the other. The manager indicates how many time steps of the PDE solver should be performed (say, N_t time steps) (lines 7-8). At this point, the different groups are working independently of each other. Once a group is done with all its tasks, it sends a signal to the manager process. Once all groups are done, the manager sends a signal to the groups to trigger the combination step. At this point, each group first combines its subset of solutions (line 13). Then, the groups combine their resulting combination solutions globally to obtain the final $u^{(c)}_{n,\boldsymbol{\tau}}$ (line 14). This combined solution can then be used as initial condition for all $u_{\mathbf{i}}$ for the next set of time steps N_t (line 17). This is repeated until some convergence criterion is met or a certain number of time steps has been carried out.

In our current implementation, all process groups comprise of the same number of processes, which makes communication easier since all process groups can use the same domain decomposition for each component grid. Therefore, when communicating across different groups, only the processes that contribute to the same geometrical region of the domain communicate with each other. This approach greatly reduces the communication overhead and has been shown to scale up to 180k cores [12].

To balance the workload, we estimate the runtime of each component grid a priori based on the number of grids points it has and its anisotropy, sort them, and distribute them among the groups (see details in [11]). In general, the larger the number of component grids per group, the easier it is to balance the workload.

Deciding how many groups to use requires some preliminary testing. If one defines a large number of groups, one can solve many tasks in parallel, which is good, but each group would have a small number of cores, limiting the size of the grids that can be solved.

Algorithm 1 The Truncated Fault-Tolerant Combination Technique in Parallel

1: Define d, n, $\boldsymbol{\tau}$, generate index set $\mathcal{I} = \bigcup_{q=0}^{d+1} \mathcal{I}_{q,\boldsymbol{\tau}}^{d,n}$ and compute coefficients $c_{\mathbf{i}}$
2: Distribute tasks among G groups with index sets $\mathcal{I}_g$, $g = 1, \ldots, G$
3: Set initial conditions $u_{\mathbf{i}} \leftarrow u(\mathbf{x}, t = 0)$, $\mathbf{i} \in \mathcal{I}$
4: Set starting time
5: **while** not converged **do**
6: **for** $g \in 1, \ldots, G$ **do in parallel**
7: **for** $\mathbf{i} \in \mathcal{I}_g$ **do**
8: $u_{\mathbf{i}} \leftarrow \text{SOLVE}(u_{\mathbf{i}}, N_t)$ ▷ Do N_t time steps of solver
9: DECIDETOKILL( ) ▷ Decide if process should die
10: **if** faults detected **then**
11: RECOVER()
12: **for** $g = 1, \ldots, G$ **do in parallel**
13: $u_{n,\boldsymbol{\tau}}^{(g)} \leftarrow \sum_{\mathbf{i} \in \mathcal{I}_g} c_{\mathbf{i}} u_{\mathbf{i}}$ ▷ Each group combines locally
14: $u_{n,\boldsymbol{\tau}}^{(c)} \leftarrow \sum_{g=1}^{G} u_{n,\boldsymbol{\tau}}^{(g)}$ ▷ All groups combine globally
15: **for** $g \in 1, \ldots, G$ **do in parallel**
16: **for** $\mathbf{i} \in \mathcal{I}_g$ **do**
17: $u_{\mathbf{i}} \leftarrow \text{SAMPLE}(u_{n,\boldsymbol{\tau}}^{(c)})$ ▷ Use combination solution as next initial value

If each component solution $u_{\mathbf{i}}$ is expensive to compute, one might be forced to define larger groups, losing some parallelism across groups.

3.2 The Fault-Tolerant Combination Technique

Now suppose faults occur, affecting one or more process groups. For simplicity, assume only one process group is affected. Its set of component solutions is therefore partially or entirely damaged. The *Fault-Tolerant Combination Technique* (FTCT) defines a way to deal with such scenarios [9]. It sets the combination coefficients of the affected component solutions to zero, $c_{\mathbf{i}} = 0$, $\mathbf{i} \in \mathcal{K}$, where $\mathcal{K} \subset \mathcal{I}$ is the subset of indices affected by faults. The rest of the combination coefficients are adapted in order to obtain the best possible combination of the component solutions unaffected by faults[2]. This alternative combination is not as good as the original, but it is still very good [9]. More importantly, we avoid creating extra checkpoints to ensure fault tolerance – the FTCT is a forward recovery technique. In some cases, however, we might choose to recompute some of the lost $u_{\mathbf{i}}$ if they are very cheap (i.e., they have comparably few grid points). The reason is that it is considerably harder to find alternative combination coefficients when these low resolution solutions go missing [9]. In these cases, the last combined solution $u_{n,\boldsymbol{\tau}}^{(c)}$ serves as a checkpoint, which is available in the memory of each process group. This will become more clear in the results section, where we will see the effect of recomputing some component solutions on the scaling results.

Evidently, if many component solutions are affected, it might not be possible to combine the other ones in a way that gives a good approximation. To ensure that there are always enough component solutions to combine, the FTCT adds a few component solutions to the original set. In particular, it adds new $u_{\mathbf{i}}$ with $|\mathbf{i}|_1 = n + |\boldsymbol{\tau}|_1 - 1$ and $|\mathbf{i}|_1 = n + |\boldsymbol{\tau}|_1 - 2$. In our 2D example with $n = 4$ and $\boldsymbol{\tau} = (0, 0)$, the component solutions added would be

$$u_{(2,1)}, \quad u_{(1,2)} \quad \text{and} \quad u_{(1,1)}.$$

These will have a combination coefficient equal to zero if no faults occur, but they might be used in the combination if some of the other solutions go missing due to faults.

3.3 Fault Simulation Layer

Faults are simulated and handled using an additional software layer that emulates a few core functionalities of ULFM [3], such as `MPI_Comm_shrink`. These allow us to detect failed communication partners and to remove them from the communicator. We chose to implement this layer in order to be able to use the native MPI installation of the *Hazel Hen* supercomputer instead of having to install ULFM, which is currently not possible. Additionally, the native installation of MPI is optimized for the system.

With our layer we can simulate process failures by calling a `KillMe()` function at given times. This function forces the calling process to go idle, returning the same type of error messages as ULFM. Our layer can, therefore, be exchanged with ULFM since the interface is the same, and we plan to do this in the future.

3.4 Fault Handling and Recovery

Faults are detected at the global communication step, during which the manager can identify failed groups. It then enters into recovery mode, first determining which component solutions had been assigned to the failed groups. It then computes the alternative combination coefficients and communicates them to the living groups, which can then combine their component solutions locally. After this step the global combination is performed, but before moving on to the next set of time steps, the manager reassigns the component solutions of failed groups to the remaining living groups. The algorithm can then continue as before.

What happens to the failed process groups? This depends on whether all processors in the group failed, or only a subset. In the past we have removed the complete process group from the communicator even if only a few of its processes were affected by faults. This is easy to implement but wastes compute resources. In our current implementation, we reuse the living ranks within a failed group (if there are any), tagging them as *spare processes* ('SP', Figure 2). These can be used to restore other groups in later iterations (Figure 3).

This is implemented by using two communicators: one *world* communicator that contains only active processes and one *spare* communicator that contains active and spare processes. Whenever a process failure occurs we execute `MPI_Comm_Shrink` on both communicators to remove the failed rank(s). Next, the manager rank checks whether there are enough spare processes to replace the failed rank(s). If this check returns true, the respective spare processes are assigned to the *world* communicator by calling `MPI_Comm_Split` on *spare*. In case there are not enough spare processes, the manager removes the failed process group and tags all

[2] Finding the optimal coefficients given the constraint $c_{\mathbf{i}} = 0$ for $\mathbf{i} \in \mathcal{K}$ is not easy (it is in fact NP-hard), so some additional considerations apply. The details are found in [9].

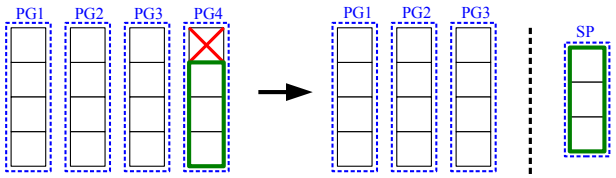

Figure 2: An example scenario with 16 processes divided into 4 process groups. After a process failure occurs in a process group (PG) the remaining MPI ranks are assigned to the group of spare processes (SP) and the failed rank is excluded.

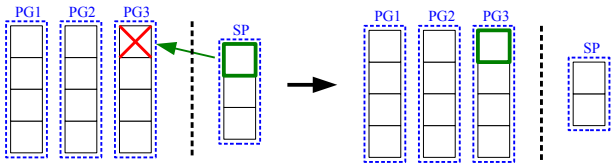

Figure 3: A process group (PG) can be restored if all failed ranks can be replaced by spare processes (SP).

the remaining processes of the group as spare processes. Again we use `MPI_Comm_Split` to remove those processes from *world*. This scheme allows for a transparent implementation of the process exchange as the application only uses the *world* communicator and does not know about *spare*. Hence, the implementation can be easily exchanged or extended in the future.

3.5 Fault Distribution

A common way to model the probability of faults affecting a system is to use a Weibull distribution function, which is given by

$$f(t;\lambda,k) = \frac{k}{\lambda}\left(\frac{t}{\lambda}\right)^{k-1} e^{-(t/\lambda)^k} \tag{5}$$

for $t \geq 0$. Here k and λ denote, respectively, the shape and scale parameter. λ is related to the mean value by $\mathrm{E}[f(t;\lambda,k)] = \lambda\Gamma(1 + 1/k)$. Hence, it can be adjusted to model environments with different failure rates.

We inject faults as proposed in [9] by drawing values from the Weibull distribution. This value is added to the initialization time of the process and marks the time the process will fail. After each computation of a component grid the process checks if the current time exceeds the failure time and calls the `KillMe()` function accordingly. This is done in the `DecideToKill()` function in Algorithm 1.

4 SIMULATION SCENARIOS AND RESULTS

4.1 Scenarios

For all our experiments, we chose an ion-temperature gradient (ITG) test case with initial condition *alm*, which is a typical initial value scenario with one species [16, Appendix A.2.1]. All simulations were run in local mode, which means that only a small part of the complete tokamak fusion reactor is simulated. We used a 3D Combination Technique of level $n = 3$ in dimensions $(z, \mu, v_\parallel)$, keeping the dimensions x and y constant (with 9 and 1 discretization points respectively). The reason for the constant y direction is that the linear and local computations in GENE are decoupled in the y direction. Therefore, every y coordinate is simulated separately. The x coordinate is constant as 9 points already offer sufficient resolution for this scenario. For global and non-linear simulations, which we plan for the future, this changes and combination in all 5 dimensions would be possible.

We present results for two scenarios: In test case A, we set the truncation parameter to $\tau = (4, 4, 4)$ in dimensions $(z, \mu, v_\parallel)$. In test case B, we use $\tau = (5, 5, 5)$. This results in 10 component grids for both scenarios. In all experiments we used 512 processes split into 4 process groups (so 128 processes per group). We combined the solutions after every iteration step and performed 6000 time-steps in total, with a time step size of 0.005 s. The shape parameter k of the Weibull distribution was set to 0.7, which accounts well for the mean-time between failures (MBFT) of a single node, including infant mortality [21]. We then analyzed the error for different failure rates λ. All tests were repeated 30 times for each value of λ to investigate the statistical properties of the errors for different failure scenarios.

4.2 Error Analysis

In Figures 4 and 5 we plot the L_2 error of the combination technique with faults compared to the reference solution of level $\mathbf{n}' = (3, 1, 8, 8, 8)$ in dimensions $(x, y, z, v_\parallel, \mu)$. Notice that the different choices of λ result in different numbers of faults – up to 117 faults for $\lambda = 10^5$. Note that the number of faults is defined as the number of iterations in which process failures occur. In case multiple processes fail in the same iteration this is counted as one fault.

The L_2 error is computed by simply taking the L_2 vector norm of the difference between the absolute values of two grids, as suggested in [10, Section 5.2.2]:

$$\sqrt{\sum_{\vec{p}\in\Omega_{\boldsymbol{n}'}} \left(\left|u_{\boldsymbol{n}'}(\vec{p})\right| - \left|u^{(c)}_{n,\boldsymbol{\tau}}(\vec{p})\right|\right)^2},$$

where $\vec{p}$ is a five-dimensional coordinate on the reference grid and $u_{\boldsymbol{n}'}(\vec{p})$ and $u^{(c)}_{n,\boldsymbol{\tau}}(\vec{p})$ the respective values of the grid at position $\vec{p}$ for the reference and the combination solution. As the combination does not contain all grid points $u_{\mathbf{n}'}(\vec{p})$, we interpolate the combined solution onto the reference grid first. The reason for taking the absolute value before calculating the difference is that the solutions may have different phase shifts in the complex plane which are not relevant for the physical interpretation. By taking the absolute value of the complex numbers, we remove the influence of the phase shift on the vector norm. In addition to this, the five-dimensional vectors $u_{\mathbf{n}'}$ and $u^{(c)}_{n,\boldsymbol{\tau}}$ are normalized to 1 – using the L_2 norm – before the norm calculation.

We can see that the error increases with the number of faults, but it still remains quite close to the reference solution. For test case B, the average error is smaller than test case B since we chose a higher truncation parameter $\boldsymbol{\tau}$ and $\boldsymbol{n}$, which results in a more accurate combination. In the worst-case scenario (test case B with $\lambda = 10^5$) the error increases by 20% on average, which, considering that each fault causes a loss of about 25% (for the first fault) to 33% (every other fault) of the component grids, this increase is

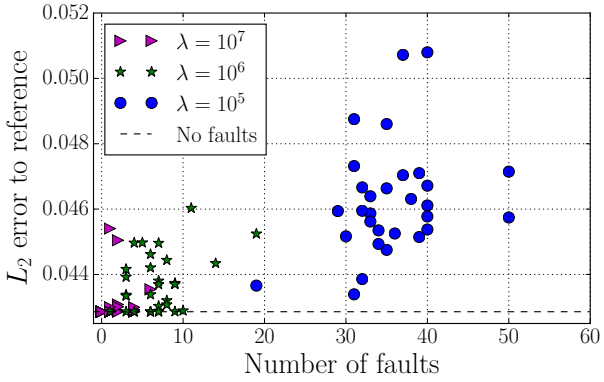

Figure 4: L_2 error of the FTCT as a function of the number of faults: test case A.

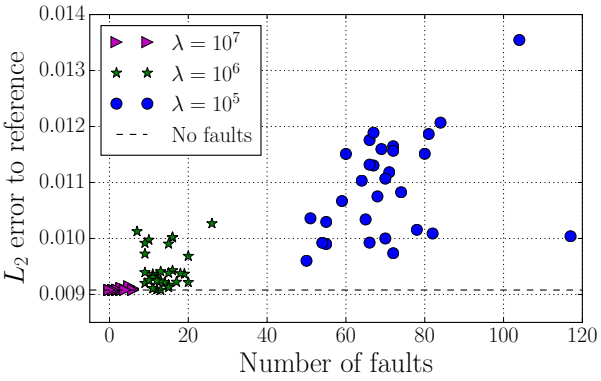

Figure 5: L_2 error of the FTCT as a function of the number of faults: test case B.

λ	f_{avg}	f_{max}	f_{min}	e_{avg}	e_{max}	e_{min}	Δe (%)
10^7	1.43	6	0	0.0431	0.0454	0.0429	0.480%
10^6	6.75	19	3	0.0437	0.0460	0.0429	1.91%
10^5	35.5	50	19	0.0463	0.0508	0.04339	7.955%

Table 1: Statistical results of the error of the FTCT for different λ in test case A. f represents the number of faults, e the L_2 error to the reference and Δe the average increase in the error compared to a simulation without faults.

λ	f_{avg}	f_{max}	f_{min}	e_{avg}	e_{max}	e_{min}	Δe (%)
10^7	2.55	6	0	0.00908	0.00915	0.00908	0.0882%
10^6	13.5	26	7	0.00945	0.0103	0.00908	4.09%
10^5	70.3	117	50	0.0109	0.0135	0.00960	20.2%

Table 2: Statistical results of the error of the FTCT for different λ in test case B.

completely tolerable. The average, maximum and minimum errors can be seen in Tables 1 and 2.

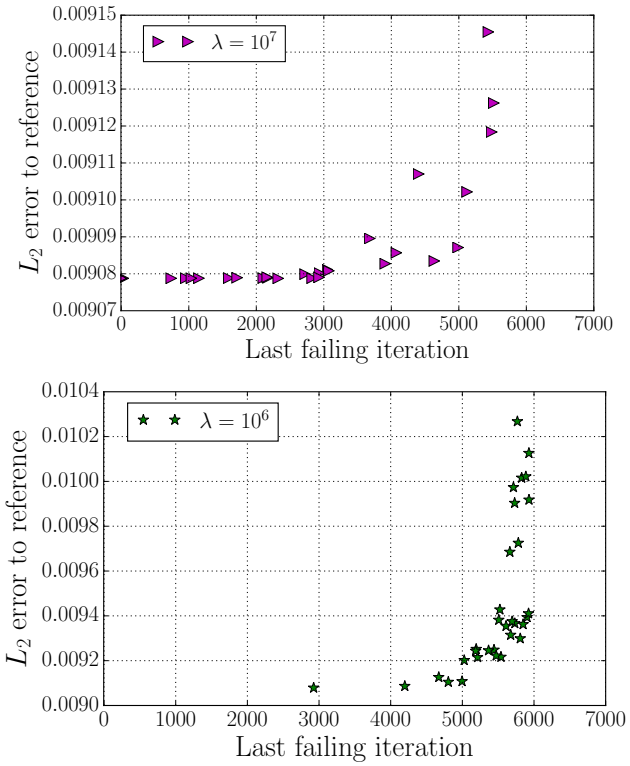

Figure 6: L_2-error of different runs of test case B compared to the last iteration with process error for $\lambda = 10^6$ (bottom) and $\lambda = 10^7$ (top) .

The iteration at which faults occur also has a significant impact on the overall error. Figure 6 shows the error of the solution compared to the last iteration in which a process failure occurred in the respective run. It seems that the later a fault occurs, the larger the resulting error. For moderate numbers of faults, this has an even larger contribution to the overall error than the total number of faults. For really high number of faults (for example, for tests with $\lambda = 10^5$ having 70 faults on average), this effect is less predominant.

4.3 Recovery overhead

Figure 7 (top) shows the times required by the different steps of the FTCT algorithm for one run of test case B with 84 faults and 512 cores. We plot the maximum runtime of the respective steps during the whole simulation. Most of the time is spent **solving** the PDE. The **combine** step takes about one sixth of the time required by the solve function[3]. The **recovery** overhead is almost negligible – 60.83 s compared to 6,780 s of the solve time (0.89% for 84 recoveries, or 0.72 s per recovered fault). This is cheaper than pure checkpointing, since it would take 1.13 s only to recompute the last time-step. Additionally, the time to **write the checkpoint** is about five times larger than one iteration step.

[3]This is an upper bound, since in these experiments we combined after every time step. In general, one can combine only after a few time steps and keep the accuracy of the combination [10, Section 5.2.2].

Naturally, the time to solve a time-step of a PDE depends on the specific application and the solver used, so it could be much smaller for other codes. However, GENE has been highly optimized over many years and is therefore an illustrative application[4]. But more importantly, the steps of the FTCT should scale.

4.4 Scaling

Figure 7 (middle) shows our scaling results for one run. Here we used different parameters for the Combination Technique to demonstrate the scalability for a large-scale simulation. For this purpose we chose $\tau = (3, 3, 3)$ and $n = 10$, which results in 185 combination grids[5]. Again, we fixed the number of discretization points in x to 513 and y to 1. We simulated 300 time-steps and combined every 100 time-steps (3 combinations in total) [6]. In all cases, each process group had 1024 processes, and we doubled the number of groups for each experiment. For the experiments with faults, we begin with 4 process groups. We plot the most expensive steps of the Combination Technique (**solve** and **combine**) with faults, as well as the cost to **recover** from one fault, always choosing the same group to fail. Those times are compared to our scaling results of a classical Combination Technique without faults [10, Section 5.2.2].

Both **solve** and **combine** are slightly more expensive in the fault scenario, since we lose one whole process group (1024 cores) when the first fault occurs, but in both cases, both steps scale. There is an additional increase in the solve time for our fault-tolerant implementation that comes from having more component grids than the classical Combination Technique, but the extra cost is very small. Similarly, the time for the combination step increases as well. The **recover** step scales well up to 16k cores and then we see a slight increase for 32k cores. Further investigations showed that the recovery time is mainly dominated by the recomputation time. For each of the runs, only a few "cheap" (very coarse) component solutions needed to be recomputed. As each of these component solutions is only solved by one process group (of fixed size in our case), the recovery time cannot scale in cases where we have more process groups than component solutions to recompute – some of the process groups are idle in this case. Furthermore, different numbers of solutions fail in the different runs, which results in different numbers of solutions that need to be recomputed. As those solutions vary in the number of points as well as in their anisotropy, recovery times vary for each experiment. This makes it difficult to predict or interpret the scaling of the recovery step. However, recomputing a single solution should scale with the number of cores in a process group if the solver – in our case GENE – still scales. Additionally, the need to recompute decreases with the number of process groups, since fewer solutions fail and therefore one can avoid recomputing in more cases.

This effect can be seen in Figure 7 (bottom) where we show the average of 3 runs for every group configuration. The average runtime of the recovery step scales well up to 32k cores. Only for 16k cores the scaling seems a little off, which is caused by one

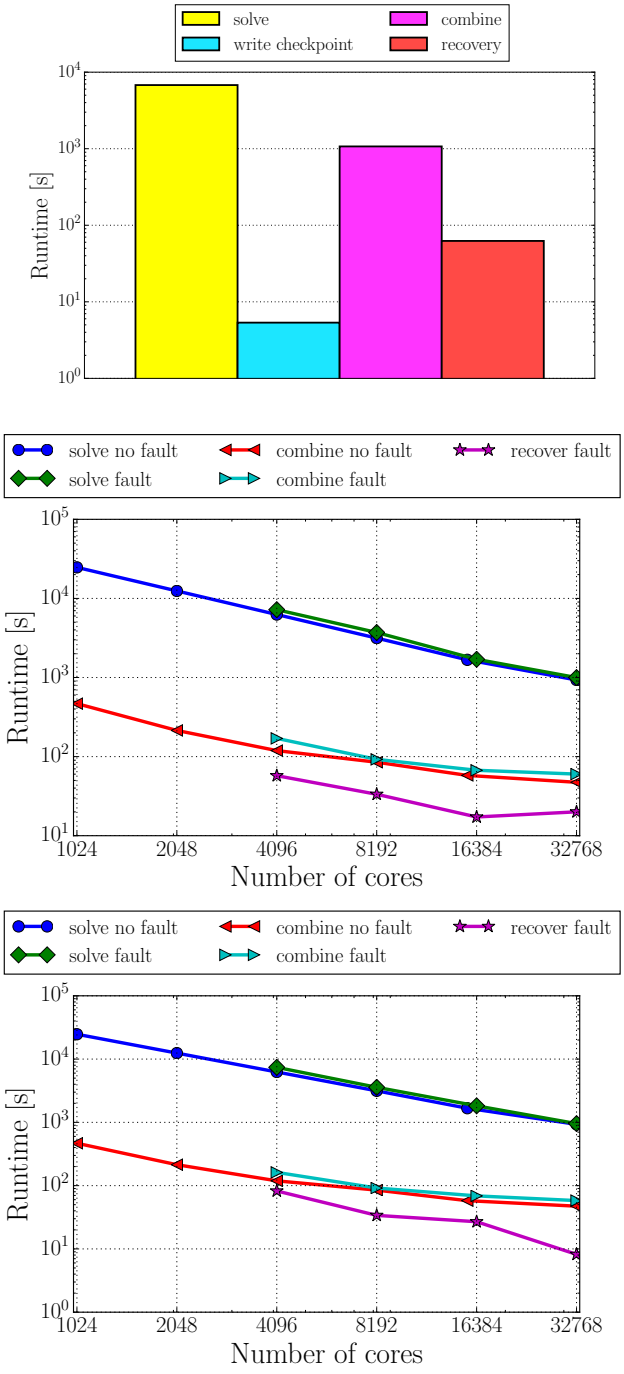

Figure 7: Top: Runtimes for the most expensive steps of the FTCT. We plot the maximum time a process spends in total for each step. Middle: Scaling results for one specific run with and without faults. In all fault scenarios, one process of the first process group fails. Bottom: Average runtimes of three scaling runs.

[4] Additionally, other more realistic simulation scenarios with GENE (e.g. nonlinear, global simulations) are much more expensive than the scenario we chose.

[5] The classical (non-fault-tolerant) Combination Technique without additional grids would have 136 component grids.

[6] The full grid solution we are approximating would have roughly $2^9 \times 1 \times 2^{13} \times 2^{13} \times 2^{13}$ grid points, which would be impossible to solve directly even on the full supercomputer.

large runtime for the recovery step in one of the three runs. The overall relative parallel efficiency is 93.61% compared to 4k cores with faults. If we compare it to 1k cores without faults, we still obtain a parallel efficiency of 76.97%. The increase in the solve time is 10.93%, and 21.37% for the combination time. It is important to remember that each run is different in the sense that the tasks are distributed non-deterministically, and so is the time needed for the recompute step. For example only in one of the three 32k test cases we had to recompute component solutions. Nevertheless, these results show that in average we can expect good scaling even with high process numbers. Finally, the recovery overhead only takes a minor portion of the overall runtime. In more realistic scenarios where faults occur very rarely and millions of time-steps (instead of 300) are simulated, this overhead would be even smaller.

5 CONCLUSIONS

In this work we showed a scalable implementation of the *Fault-Tolerant Combination Technique* to solve higher-dimensional PDEs, and in particular the gyrokinetic Vlasov equations. The Combination Technique allows us to increase the discretization resolution of a simulation at lower cost than on a full grid, and it is also fault tolerant. We simulated various fault scenarios with a high number of faults and showed that the error remains small. Even in extreme scenarios with up to 117 independently failing ranks the error increased in average only by 20%. In addition to the number of faults, the time at which faults occur also has a significant impact. For GENE we noticed that faults at the end of the simulation result in less accurate solutions compared to faults at earlier time-steps.

We also showed that the overhead of the recovery method is smaller than the cost of performing a single iteration in a test-case with 512 cores. This is much cheaper than pure checkpointing – writing a single checkpoint took about five times as long. For a large test-case with 185 component grids and one failing rank we observed good scalability on up to 32k cores. Moreover, the runtime overhead compared to a non-fault-tolerant Combination Technique is small. In particular, for all test-cases the recovery overhead, i.e. the time for reorganizing communicators, redistributing and recomputing of component grids, remained small compared to the solver runtime and therefore has only a minor effect on the scalability. We conclude that the fault-tolerant Combination Technique is a good method to achieve reliable results even in environments with high failure rates.

6 RELATED WORK

The FTCT has been studied in several works. In [9] Harding et al. investigated the statistical properties of the FTCT. However, only the influence of very few faults is examined and only a simple advection problem is investigated.

A parallel implementation of the FTCT with GENE is presented in [2]. The authors use ULFM-MPI and can tolerate real system faults. In contrast to our implementation, the combination is performed on the target full grids, which increases the memory overhead significantly. In [1], the framework is also applied to Lattice Boltzmann and SFI applications, with a scaling analysis up to 3k cores. In [10, Chapter 3.6] further comparison to our framework concerning the load balancing scheme and the combination structure can be found.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) through the Priority Programme *Software for Exascale Computing* (SPPEXA).

REFERENCES

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

[2] Md Mohsin Ali, Peter E. Strazdins, Brendan Harding, Markus Hegland, and Jay W Larson. 2015. A Fault-Tolerant Gyrokinetic Plasma Application using the Sparse Grid Combination Technique. In *Proceedings of the 2015 International Conference on High Performance Computing & Simulation (HPCS 2015)*. Amsterdam, The Netherlands, 499–507.

[3] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254.

[4] Hans-Joachim Bungartz and Michael Griebel. 2004. Sparse grids. *Acta Numerica* 13 (2004), 147–269. https://doi.org/10.1017/S0962492904000182

[5] Franck Cappello et al. 2014. Toward Exascale Resilience: 2014 update. *Supercomputing frontiers and innovations* 1, 1 (2014).

[6] Jochen Garcke and Michael Griebel. 2000. On the Computation of the Eigenproblems of Hydrogen and Helium in Strong Magnetic and Electric Fields with the Sparse Grid Combination Technique. *J. Comput. Phys.* 165, 2 (2000), 694–716.

[7] Michael Griebel, Walter Huber, and Christoph Zenger. 1996. Numerical Turbulence Simulation On A Parallel Computer Using The Combination Method. In *Flow simulation on high performance computers II*. 34–47.

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

[9] Brendan Harding et al. 2015. Fault Tolerant Computation with the Sparse Grid Combination Technique. *SIAM Journal on Scient. Comp.* 37, 3 (2015), C331–C353.

[10] Mario Heene. 2017. *A massively parallel combination technique for the solution of high-dimensional PDEs.* Ph.D. Dissertation. University of Stuttgart.

[11] Mario Heene, Christoph Kowitz, and Dirk Pflüger. 2013. Load Balancing for Massively Parallel Computations with the Sparse Grid Combination Technique.. In *Parallel Computing: Accelerating Comp. Science and Eng.* 574–583.

[12] Mario Heene, Alfredo Parra Hinojosa, Hans-Joachim Bungartz, and Dirk Pflüger. 2016. A Massively-Parallel, Fault-Tolerant Solver for High-Dimensional PDEs. Euro-Par. Accepted.

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

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

[15] Frank Jenko et al. 2000. Electron temperature gradient driven turbulence. *Physics of Plasmas (1994-present)* 7, 5 (2000), 1904–1910. http://www.genecode.org/

[16] Christoph Kowitz. 2016. *Applying the Sparse Grid Combination Technique in Linear Gyrokinetics.* Dissertation. Technische Universität München, München.

[17] Bernd Mohr and Wolfgang Frings. 2010. *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.

[18] Alfredo Parra Hinojosa, Christoph Kowitz, Mario Heene, Dirk Pflüger, and H-J Bungartz. 2015. Towards a fault-tolerant, scalable implementation of GENE. In *Recent Trends in Computational Engineering-CE2014*. Springer, 47–65.

[19] Dirk Pflüger, Hans-Joachim Bungartz, Michael Griebel, Frank Jenko, Tilman Dannert, Mario Heene, Christoph Kowitz, Alfredo Parra Hinojosa, and Peter Zaspel. 2014. EXAHD: an exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *European Conference on Parallel Processing*. Springer, 565–576.

[20] Yves Robert. 2016. An overview of fault-tolerant techniques for HPC. (2016). http://graal.ens-lyon.fr/~yrobert/europar16.pdf Euro-Par.

[21] Bianca Schroeder and Garth Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (Oct 2010), 337–350. https://doi.org/10.1109/TDSC.2009.4

[22] 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. 2014. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* 28 (2014), 129–173. Issue 2.