Doubt and Redundancy Kill Soft Errors—Towards Detection and Correction of Silent Data Corruption in Task-based Numerical Software

Philipp Samfass
Department of Informatics
Technical University of Munich
Garching, Germany
samfass@in.tum.de

Tobias Weinzierl
Computer Science
Durham University
Durham, United Kingdom
tobias.weinzierl@durham.ac.uk

Anne Reinarz
Computer Science
Durham University
Durham, United Kingdom
anne.k.reinarz@durham.ac.uk

Michael Bader
Department of Informatics
Technical University of Munich
Garching, Germany
bader@in.tum.de

***Abstract*—Resilient algorithms in high-performance computing are subject to rigorous non-functional constraints. Resiliency must not increase the runtime, memory footprint or I/O demands too significantly. We propose a task-based soft error detection scheme that relies on error criteria per task outcome. They formalise how "dubious" an outcome is, i.e. how likely it contains an error. Our whole simulation is replicated once, forming two teams of MPI ranks that share their task results. Thus, ideally each team handles only around half of the workload. If a task yields large error criteria values, i.e. is dubious, we compute the task redundantly and compare the outcomes. Whenever they disagree, the task result with a lower error likeliness is accepted. We obtain a self-healing, resilient algorithm which can compensate silent floating-point errors without a significant performance, I/O or memory footprint penalty. Case studies however suggest that a careful, domain-specific tailoring of the error criteria remains essential.**

***Index Terms*—soft errors, detection, correction, fault tolerance, fault resilience**

I. Introduction

Without a revolutionary hardware re-design, a massive further reduction of clock frequency, or an increasing power budget accommodating hardware failure correction, we have to assume that exascale machines will fail frequently compared to today's machines [1], [2]. Empirical data leads us to expect a linear correlation between the system size and the failure rate [3]. The mean time between failures (MTBF) will shrink. Simulation codes thus have to improve their resiliency. In particular, they have to become able to identify machine errors and to handle them.

Resilient codes traditionally run through two phases: First, they spot the errors which can either materialise in machine part failures (hard errors) or wrong results (soft errors). Soft errors are easy to spot if they materialise in exceptional values ($\pm\infty$, NaN, e.g.). In numerical simulations it is tricky to identify them otherwise, as we typically work with approximations. We have to define how far off from a result is considered to be a soft error, while we might struggle to determine this difference given that we typically do not know the exact solution. The "simplest" approach is to not detect soft errors at all, but to rely on an iterative algorithm, and iterate as long as soft errors continue to pollute the outcome [4]–[8]. To actually detect errors, codes can rely on algorithmic a-posteriori checks [9], or they can run multiple redundant computations and compare their outcomes [10]–[12]. The last approach requires codes to run the same calculation on different machines or machine parts: at least twice to detect errors or even three times to label the "correct" solution via a majority vote. In a second phase, resilient codes have to fix the wrong data. Here, three strategies are on the table: Codes can rely on algorithmic fixes—checksums for example allow them to reconstruct results, while the aforementioned iterative algorithms have the correction built in. They can rely on a rollback to a previous snapshot which has been declared valid, or they can swap in a redundant data set from a valid, redundant calculation. All strategies assume that soft errors arise sporadically and the machine remains, in principle, intact.

In a supercomputing context, these resiliency strategies need to satisfy important non-functional requirements. A resiliency strategy **should not introduce** significant

(i) additional synchronisation between otherwise independent calculations: synchronisation hinders scalability;
(ii) additional communication bandwidth or latency: network bandwidth and responsiveness are precious resources that quickly develop into a bottleneck if stressed too much;
(iii) I/O needs: I/O operations are traditionally by magnitudes slower than compute and communication tasks and thus slow down the calculation;
(iv) additional memory footprint: supercomputing codes typically try to increase the memory usage per node already to avoid strong scaling stagnation effects.

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the grant agreement No 823844 (project ChEESE). Tobias acknowledges the support from the ExCALIBUR Phase I grant ExaClaw (EP/V00154X/1) and ExCALIBUR's cross-cutting project EX20-9 Exposing Parallelism: Task Parallelism (grant ESA 10 CDEL). The Exascale Computing ALgorithms & Infrastructures Benefiting UK Research (ExCALIBUR) programme is supported by the UKRI Strategic Priorities Fund. The programme is co-delivered by the Met Office on behalf of PSREs and EPSRC on behalf of UKRI partners, NERC, MRC and STFC.

Our work starts from the observation that many simulations decompose their work into work items (tasks over mesh cells, e.g.) which allow us to define strong or weak confidence metrics on the outcome of these items: Negative mass density or NaNs unambiguously flag wrong data. Sudden changes in eigenvalues feeding into an admissible time step size or sudden oscillations make outcomes dubious—they might be actually correct but the changes might also stem from a soft error. In the absence of an algorithmic postprocessing step which cures and eliminates errors (cmp. limiter-based techniques [9] in our case), we propose to run our task-based simulation twice in parallel. As long as tasks yield outcomes where we are confident that they are reasonable, we make the two replicas share their outcomes. Effectively, each replica only computes half of the work and relies on its counterpart to compute the remainder [13]. As soon as a task outcome is dubious, our code waits for the counterpart simulation to compute the outcome redundantly, and we then compare the results.

Soft errors, i.e. silent data corruption, are notoriously difficult to spot if there is no strong and immediate validation, such as a hash value that reports data corruption reliably and immediately after the error has arisen. In this case, only comparisons to redundant results with majority vote help to identify and fix them. Such an approach is infeasible in HPC, as it synchronises, triples the compute workload, and requires resources to host checkpoints. Due to task-based error criteria, we can offer soft error detection without any immediate synchronisation. We work on a per-task level, asynchronously, and run the bit-wise redundancy checks only upon demand. The error criteria also allow us to replace a majority vote with a confidence vote, while result sharing among trusted outcomes reduces the cost of the redundant computations. In many cases, silent data corruption can immediately be corrected, while the algorithm also provides evidence when rollback-and-recompute becomes mandatory.

Our strategy towards saving the cost of replication by sharing outcomes is different from other work which runs replicas at a reduced execution rate for power savings [14]. In contrast to task replay techniques in asynchronous many-task runtime systems [15], our approach has reduced algorithmic latency—the replicated computation runs in parallel—yet has a higher memory footprint to store results temporarily until they are approved. It is also able to recover from hard errors (not shown), as we work with real rank redundancy.

Our concepts are illustrated within the wave equation solver ExaHyPE [16] which uses an explicit time stepping scheme to tackle the underlying hyperbolic equations and which relies on an independent library called teaMPI [13] for transparent replication of ranks and for task outcome sharing between those ranks. Explicit time stepping schemes are notoriously difficult to equip with resiliency, since they typically operate with large data sets and are computationally demanding, while the absence of an iterative solver step or diffusion implies that errors spread out, propagate and pollute the outcome if they are not immediately corrected. Since we neither need a tailored error correction scheme [9], nor checkpointing, nor multiple redundant computations [12], our ideas improve significantly upon the state-of-the-art how to deliver resilient simulation codes, plus they are of broad applicability and relevance: As long as a solver's algorithmics can be broken down into small tasks and credibility metrics can be defined, our algorithmic ideas are applicable.

The paper is organised as follows: We present our key ideas and terminology in Section II as an abstract algorithmic framework. In Section III, we introduce our wave equation solver and highlight how the framework's concept of error criteria is realised for our application. This potential impact on a wide range of applications is supported by numerical results (Section V) which we present after Section IV's discussion of implementation details. With a realisation sketch and experimental results at hand, we can classify and contextualise our approach and thus highlight its broader impact (Section VI). A brief summary and outlook in Section VII close the discussion.

II. Algorithmic framework

Let an algorithm consist of tasks τ_i that take some input θ_i and yield output $y_i = \tau_i(\theta_i)$. Some tasks have temporal dependencies, i.e. $\tau_i \sqsubset \tau_j$: τ_i feeds into τ_j and thus has to terminate before τ_j starts. A task scheduler exploits the freedom for any task pair $\tau_i, \tau_j : \tau_i \not\sqsubset \tau_j \wedge \tau_j \not\sqsubset \tau_i$ to deploy such tasks τ_i and τ_j concurrently to the available compute cores as the tasks are *independent*.

A *hard error* within a task makes our system crash or prevents the task from terminating. A timeout can detect the latter situation so hard errors can be reliably detected. We focus on *silent errors* which make a task yield $\tilde{y}_i$ instead of the correct $y_i^{\text{(correct)}} = \tau_i(\theta_i)$. As our numerical simulations work with floating point approximations where the outcome of a task is not absolutely deterministic due to register data transfers or operation reordering, we characterise a silent error through $|\tilde{y}_i - y_i^{\text{(correct)}}| > tol_y$ ($|.|$ being a suitable norm). For simulations, we do not know $y_i^{\text{(correct)}}$ for a task prior to its execution and thus struggle to characterise a soft error unless an error yields NaNs or unreasonable data. The latter term remains to be defined.

Let a *team* be the set of MPI ranks that are used for a program run. If we run a simulation completely redundantly, we have two teams A and B. Both issue the same tasks τ_i^A and τ_i^B. The two team schedulers however might deliver different task execution sequences for tasks without a total order. As long as both teams yield correct data, i.e. do not suffer from soft errors, $|y_i^A - y_i^B| \leq tol_y$. tol_y depends on the machine precision and error propagation during the task, so usually encodes a relative error in IEEE floating point precision with a fixed number of significant bits.

An *error criterion* is a hash function $f\colon y_i \mapsto [0, \infty]$, where $f(y_i) \leq tol_f$ indicates that there is no reason to assume that a soft error has occurred during the calculation. $f(y_i) = \infty$ signals that something went wrong. Any value in-between highlights that a silent error might have crept into the result. f thus quantifies to which degree the task outcome is dubious.

Due to the numeric nature, there is however no guarantee that $f(y_i) = 0$ implies that no error has occurred and there is no direct correlation between tol_y and tol_f. $f(y_i^A) > f(y_i^B)$ does not necessarily mean that y_i^A is wrong. However, $f(y_i^A) \gg f(y_i^B)$ suggests that $|y_i^A - y_i^{\text{(correct)}}| > |y_i^B - y_i^{\text{(correct)}}|$. As silent errors are very unlikely to affect both outcomes, the one with the lower f-value is likely correct. Depending on the context, f evaluates an absolute value of y_i or takes historical data such as previous values of a numerical solution into account and calibrates y_i accordingly.

We combine multiple error criteria f_k to an *error indicator* $\phi(y_i)$, which can yield boolean or numerical values. When using booleans, we define $\phi(y_i)$ via a logical predicate, such as $\exists f_k : f_k(y_i) > tol_{f_k}$. When using numerical values, we may again use a tolerance $\phi(y_i) > tol_\phi$ to indicate dubious results. Different error criteria might exist for different tasks. To streamline our notation, we focus on one task type τ only. Let $f_1, f_2, \ldots$ then denote different criteria for task type τ, with tolerances $tol_1, tol_2, \ldots$.

Our algorithmic framework relies on few key ingredients:

- There are two teams A and B. Their scheduler is initialised with a bias seed: If two tasks τ_i and τ_j are independent and if A's scheduler gives τ_i a high priority and τ_j a low priority, then B's scheduler gives τ_j a high priority and τ_i a low one. This bias manifests non-deterministically in different task completion sequences.
- Whenever a task completes, we can compute its error criteria f_k.
- Each team has a *local cache* of task outcomes; and hence of their error criterion values. We use this cache to inform the other team whenever we have completed a task earlier than the team counterpart, and we use the cache also to temporarily store local task results if we are not sure whether they have been corrupted.

The algorithm then reads as follows (cf. Figure 1):

1) If a task is to be executed and the task outcome is not yet in the cache, compute it locally.
2) If a local task computation yields error criterion values which indicate that the result is trustworthy, share the task outcome with the counterpart in the other team. Continue with the computation.
3) If a local task computation yields error criterion values which indicate that the result might be compromised, share this outcome, too, but place the local result in the local cache for the time being. Do not accept the task outcome yet. Let the computing core progress with another task and check the outcome later.
4) If a task is to be executed, if the task outcome is already in the local cache, i.e. has been sent in by the other team, and if this incoming data has error criteria which suggest that these data are trustworthy, skip the local computation and accept the data sent in as task result.
5) If a task is to be executed, the task outcome is in the local cache and is flagged as dubious, run the calculation locally and share the outcome. Check the local outcome if it is not trustworthy.

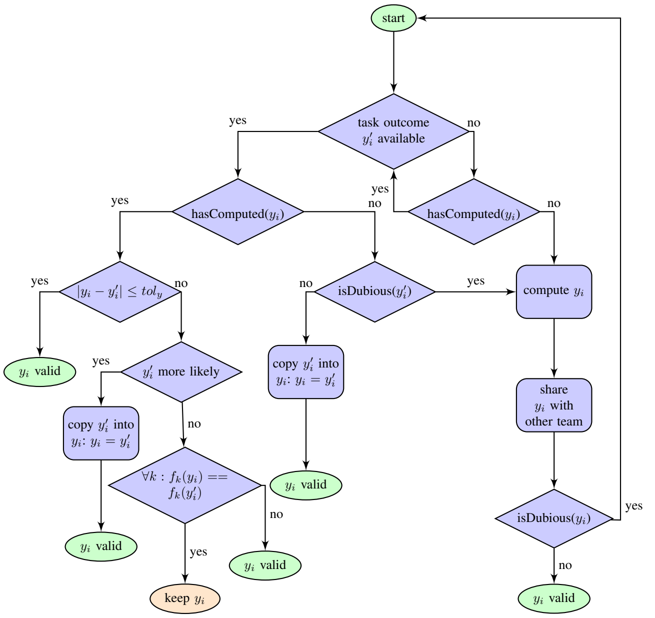

Fig. 1. Control flow to compute, share, check and correct the outcomes y_i of a task execution. y_i' denotes a task outcome from another team.

The scheme is complemented by some garbage collection that discards task results that are no longer required. This happens for tasks that drop in as problematic while the local computation yielded a result with low error criteria values, or two task outcomes that cross in the network.

As long as all tasks yield trustworthy results, the above scheme with the biased scheduler shares the actual compute workload equally among the two teams. As soon as a dubious task outcome arises on either team, the task result is backed up in the local cache. We then have to match two outcomes for the same task within the cache based upon the error criteria:

- If one task outcome has high error criteria values and the other task outcome has low error criteria values, we assume that the latter is the valid result and that the former includes some silent error. We accept and continue with the valid data.
- If both task outcomes have the same error criteria values and $|y_i^A - y_i^B| \le tol_y$, our algorithm had become over-dubious: It turned out that the result is “kind of surprising” but valid.
- If both task outcomes have the same error criteria values but disagree (i.e., $|y_i^A - y_i^B| > tol_y$), we have spotted a silent error and need to moderate (e.g., keep the local y_i).
- If both task outcomes have error criteria values of ∞, we have spotted a hard or intrinsic algorithmic error. The code has to terminate immediately, and it is up to the user to decide whether to alter the setup or to restart from the latest checkpoint.

It is straightforward to supplement the algorithmic framework with some timeouts: If the redundant task's outcome does not arrive within a certain timeframe for dubious local results, or

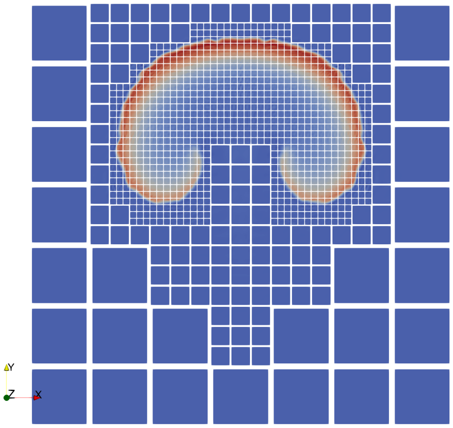

Fig. 2. Two-dimensional cut through our three-dimensional benchmark setup: We solve the compressible Navier Stokes equations to simulate a rising cloud on a dynamically adaptive Cartesian grid [17].

if one team sends out task outcomes but does not receive any outcomes from the other over a longer period, it is reasonable to assume that the counterpart team suffers from a hard error. In this case, the healthy team can still checkpoint. Checkpoint-restart thus integrates seamlessly into our algorithmic framework. As hard errors are out-of-scope in the present work, we neglect further timeout discussions from hereon.

III. Demonstrator and Benchmark Scenario

We realise our algorithmic ideas within the ExaHyPE engine [16], relying on previous work to share task outcomes between teams using the teaMPI library [13]. ExaHyPE provides a parallel software infrastructure to solve hyperbolic partial differential equations (PDEs) in their first-order form, e.g.:

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q, \nabla Q) = S(Q). \tag{1}$$

We solve for a vector of unknowns $Q(x,t)$ in time and space. The formulation requires a flux tensor $F(Q)$ and a source term $S(Q)$ plus possibly further terms omitted here. ExaHyPE's numerics are based upon the ADER-DG scheme [18], which discretizes the PDE per element in space and in time with high order polynomials and develops the solution in time with a predictor-corrector scheme. The computational domain is discretised by a dynamically adaptive Cartesian mesh [17]. Each mesh cell (cube) carries a solution polynomial. Per time-step, we run through three sub-steps:

1) A cell-local solve yields the so-called *space-time predictor* $\hat{Q}(x,t)$, a high-order polynomial approximation of the solution within the given cell. This solution neglects the influence of neighbour cells. As we compute it per mesh cube, the space-time predictors can be executed independently over the mesh.
2) A Riemann solve at each cube face in space and time computes the numerical fluxes across the faces which are induced by $\hat{Q}$, and thus captures the influence of adjacent cells upon each other.
3) A corrector step combines the space-time predictor $\hat{Q}$ with the result from the Riemann solves into a new solution $Q(x, t + \Delta t)$. Corrections are again computed per cell and hence can run independently.

Our code combines non-overlapping domain decomposition along the Peano space-filling curve with task-based parallelisation. The domain decomposition is realised via MPI. After each space-time prediction step, we have to exchange the domain boundary data to allow all ranks to compute all relevant Riemann problems. We have one data exchange per time step which materialises in one MPI message burst. Further data exchange such as a global reduction to determine an admissible time step size is negligible w.r.t. bandwidth.

The tasking classifies all cells per rank into cells that are adjacent to mesh resolution transitions or to the MPI boundary and cells that can reside within the rank's domain interior and thus can be handled with lower priority. This enclave tasking [19] implies that MPI data transfer and space-time predictor computations overlap, and that we obtain good per-node scaling [20]. The space-time prediction is the computationally dominant task type in ExaHyPE [20]. We thus focus on this task τ in the context of our resiliency work.

teaMPI [13] is a wrapper around MPI which plugs into MPI's tools interface PMPI. The teaMPI wrapper hides that we operate with two replicated teams, by using split communicators such that ExaHyPE is unaware of the redundancy. ExaHyPE leverages teaMPI's interface for task outcome caching, for communication of task outcomes and for querying whether an outcome is available through a replica computation.

In our replica/teaMPI mindset, task outcome caches synchronise with their MPI rank counterpart only, as both teams A and B employ exactly the same non-overlapping domain decomposition. If a task τ_i^A is spawned on rank r within team A, the same task will be spawned by rank r in team B eventually. No data exchange between different rank numbers within the two teams is necessary.

A. Test setup

Our test benchmark solves the three-dimensional compressible Navier Stokes equations

$$\frac{\partial}{\partial t}\underbrace{\begin{pmatrix} \rho \\ \rho v \\ \rho E \end{pmatrix}}_{=Q} + \nabla \cdot F(Q, \nabla Q) = \underbrace{\begin{pmatrix} 0 \\ -gk\rho \\ 0 \end{pmatrix}}_{=S(Q)}, \tag{2}$$

where

$$F(Q, \nabla Q) = \begin{pmatrix} \rho v \\ v \otimes \rho v + pI + \sigma \\ v \cdot ((\rho E + p)I + \sigma) - \kappa \nabla T \end{pmatrix}.$$

Here, ρ denotes the density, ρv the momentum and ρE the energy density. The pressure is given by p and the temperature by T. The constant κ is used to model diffusion of the temperature in the term $\kappa \nabla T$. A stress tensor $\sigma = \sigma(Q, \nabla Q)$ accounts for viscosity. In the source term, k denotes the unit vector in vertical direction and g the gravity of Earth.

To solve the PDE system (2), we use the ExaHyPE-based ADER-DG solver by Krenz et al. [21] to simulate a rising warm air bubble (see Fig. 2). This scenario reproduces the setup from [22], where a perturbation in a potential temperature field propagates over a background state that is in hydrostatic balance.

B. Error criteria

ExaHyPE realises an explicit time stepping scheme where the space-time predictor $y_i = \tau_i(\theta_i)$ for each cell maps a polynomial onto an extrapolated polynomial. Any error that is introduced to the solution perturbs the outcome polynomial. We may assume that localised errors affecting individual sample points within our Gauss-Legendre ansatz decrease the smoothness of the overall polynomial. At the same time, (2) prevents certain solution values such as negative densities:

a) Arithmetic corruption: If $y_i = \tau(\theta_i)$ contains NaNs, the data y_i has been compromised. We can employ a simple checksum/reduction to identify these cases. Let $f_{\text{NaN}}(y_i) = \infty$ if y_i holds NaNs. In this case, the task outcome is invalid. Otherwise, $f_{\text{NaN}}(y_i) = 0$. We do not employ a tolerance tol_{NaN} in this case or may work with any $0 < tol_{\text{NaN}}$.

b) Physical corruption: Our application setup solves conservation laws subject to certain plausibility checks. We call them physical admissibility (PA) checks. The solver tracks the density and potential temperature of the solution as primary variables over the mesh. Both are subject to the PDE and have to be non-negative. The physical admissibility criterion furthermore can derive a pressure from all of the primary quantities plus some material parameters. This pressure serves as further admissibility criterion, as it always has to be positive, too. Let $f_{\text{PA}}(y_i) = \infty$ if y_i violates the physical admissibility check. Otherwise, $f_{\text{PA}}(y_i) = 0$. Like $f_{\text{NaN}}(y_i)$, $f_{\text{PA}}(y_i)$ is a boolean label, too.

c) Dubious time step size changes: ExaHyPE relies on adaptive time stepping, i.e. the time step size Δt is not prescribed, but depends on the largest eigenvalue of the flux (in the above example, both the largest eigenvalue of the flux and the viscous flux), the polynomial order and the mesh size. It is chosen thus to fulfil the CFL condition, i.e. follows the speed information spreads through the grid.

We store the time step size Δt_i per cell and thus can determine how significant the time step size per cell changes from one time step to the other. If we employ global adaptive time stepping, the overall time step size results from a reduction over cell-local time step sizes. For the resiliency strategy, solely the local data are of interest.

Let $f_{\Delta t}(y_i) = |\Delta t_i^{\text{new}} - \Delta t_i^{\text{old}}| / \Delta t_i^{\text{old}}$. $f_{\Delta t}(y_i)$ accepts that the time step size per cell changes—as waves enter or leave the cell—but doubts the task result if this change is, relative to the previous time step, significant. The criterion indirectly relates the eigenvalues of the PDE over a cell to historic data.

d) Solution smoothness evolution: $\frac{\partial^2}{\partial x_d^2}$ denotes the second partial derivative operator in direction d. As we work with element-wise polynomial solutions in ADER-DG, it is straightforward to determine the second derivatives over the sample points ξ_n per cell. Let

$$f_{\text{Der},d}(y_i) = \frac{1}{N} \sum_{\xi_n} \frac{\left| \frac{\partial^2}{\partial x_d^2} y_i^{\text{new}}(\xi_n) - \frac{\partial^2}{\partial x_d^2} y_i^{\text{old}}(\xi_n) \right|}{\left| \frac{\partial^2}{\partial x_d^2} y_i^{\text{old}}(\xi_n) \right|}.$$

This error metric computes to which extent a newly computed task outcome increases the maximum second derivative compared to its previous value. It considers all directions separately. We evaluate the second derivative at each sample point ξ_n of the polynomial's Lagrangian representation, and sum up all obtained values per direction to obtain a single reduced value $f_{\text{Der}}(y_i) := \sum_d f_{\text{Der},d}(y_i)$. While small changes of the maximum second derivative are natural as waves propagate, very large values of $f_{\text{Der}}(y_i)$ are suspicious. They flag drastic changes of the solution smoothness. This can happen for non-linear equations due to wave stiffening, yet is rare.

e) Combining multiple error criteria: A free choice of $tol_{\Delta t}$ and tol_{Der} allows the user to incorporate domain-specific knowledge ("is the solution usually smooth or are shocks/steep gradients typical" for example) and facilitates a balancing between sensitivity and speed.

We support rigorous or lazy evaluation of the error criteria. In the rigorous variant, all error criteria are evaluated. If any criterion is violated, the outcome is seen as dubious and needs to be checked further. We obtain $\phi(y_i) = (f_{\text{NaN}}(y_i) > 0) \vee (f_{\text{PA}}(y_i) > 0) \vee (f_{\text{Der}}(y_i) > tol_{\text{Der}}) \vee (f_{\Delta t}(y_i) > tol_{\Delta t})$ as a boolean dubiosity error indicator. Here, $\vee$ denotes a strict logical OR. It ensures that all error indicator values are available for comparisons with a matching task outcome from another team for $\phi(y_i) = 1$. In the lazy variant, we first evaluate the computationally cheap error criteria $f_{\text{NaN}}, f_{\text{PA}}$ and $f_{\Delta t}$. Only if one of these pre-filtering criteria is violated, the derivative error criterion f_{Der} is evaluated. The task outcome is dubious if and only if it violates one of the pre-filtering criteria *and* the derivatives criterion, i.e. $\phi(y_i) = [(f_{\text{NaN}}(y_i) > 0) \vee (f_{\text{PA}}(y_i) > 0) \vee (f_{\Delta t}(y_i) > tol_{\Delta t})] \wedge\wedge (f_{\text{Der}}(y_i) > tol_{\text{Der}})$. Here, $\wedge\wedge$ denotes a *non-strict* logical AND. If the pre-filtering criteria are non-dubious, we formally assume $f_{\text{Der}} = 0$.

Once two task outcomes' criteria differ, we have to decide which outcome is erroneous. We base this decision on comparisons of the individual error criterion values, i.e., y_i^A is more likely than y_i^B according to f_k if $f_k(y_i^A) < f_k(y_i^B)$. We cascade these comparisons according to the order $f_{\text{NaN}}, f_{\text{PA}}, f_{\text{DER}}$ and $f_{\Delta t}$ non-strictly, i.e., we stop as soon as one f_k has flagged y_i as more likely. This results in a prioritized evaluation. Criteria checked earlier are not allowed to contradict subsequent criteria for marking an outcome y_i^A as more likely, i.e., we demand that $f_k(y_i^A) == f_k(y_i^B)$ for these criteria. This can result in situations in which we cannot decide upon the erroneous outcome. In these cases, we keep the local outcome and emit a warning indicating that a silent error could not be corrected. Further error recovery measures could then be activated to avoid error propagation.

IV. Implementation

A. Selection of tasks

The space-time predictor tasks are reasonable to hook in an error indicator, as they match the non-functional requirements that we identified for the overall resilient algorithm:

(i) The cost to compute a task of interest has to be high compared to the evaluation cost of the error criteria. An upper threshold for the f-cost is the computational time to compute the core task τ itself—otherwise, the effective cost per task doubles, and even if the two teams perfectly share their outcomes, the absolute time-to-solution remains invariant. In practice, we expect to see at least some runtime savings in return for investing twice as many resources and the cost of the f-evaluations thus has to be significantly smaller. For our space-time predictor, we observe that the cost for the dubiosity checks are significantly smaller than the τ computation, as the space-time predictor solves a dense non-linear problem.

(ii) The number of ready tasks that can be shared between teams has to be high. Only if the runtime is dominated by phases when a lot of ready tasks linger in the system, we can shuffle their execution order and hence profit from task outcome sharing, and exchange task outcomes while other tasks are still computed. For ADER-DG, the first phase per time step, which issues embarrassingly parallel space-time prediction tasks only, is a perfect fit to this requirement. Previous work of ours has demonstrated that we can shuffle the execution order of these tasks slightly by assigning task priorities, and obtain reasonable task sharing ratios as long as the tasks remain uncorrupted and we can assume that all incoming task outcomes are valid [13].

(iii) Tasks must, on an academic notion of the task concept, be atomic and final: They must not have any immediate side effects, and it must be possible to delay re-using outcomes in follow-up computations. Furthermore, tasks are not allowed to interrupt or spawn further tasks. Each STP task can run independently to other STP tasks as it accesses only element-local data. An STP's result feeds into Riemann solves at the adjacent cell-faces of an element for computing the numerical flux from and to its neighbours, but it does not directly yield further tasks. Instead, we wait for the corrections to be finished before we issue the next type of tasks (Riemann solves).

(iv) The memory footprint of a task's output must be relatively small. We have to share task outcomes between rank pairs from different teams, and we have to cache task results locally whenever a task result drops in or our local computations suggests that some dubiosity and consistency checks become necessary. Furthermore, only small footprints ensure that we can transfer output quickly via MPI and the task result sharing does not introduce interconnect congestion. Our preliminary work [13], which we use as code base here, has demonstrated that task outcomes can be shared in a timely manner as long as we take special care regarding the MPI progression. However, the present approach still runs risk to double the effective memory footprint per rank.

(v) The likelihood that silent data corruption affects the task outcome directly has to be high. Our approach relies on local checks and immediate correction of corrupted task outcomes. STP tasks account for most of the consumed CPU time during an ExaHyPE run [9], [20], making them very likely to be affected by silent data corruption. Previous work using a simplified oscillation analysis (min/max condition) in combination with the physical admissibility criterion furthermore suggests that these two criteria can identify up to 60% of the significant silent data corruptions within STPs for the Euler and Einstein equations [9]. However, it remains to be validated experimentally to which degree these insights carry over to our lazy dubiosity checks, apply to our application domain, and how a selection of dubiosity tolerances affects both the runtime and the error detection rates.

While the space-time predicator tasks τ are responsible for the bulk of the compute cost, they are not the only tasks within our system. Other, cheap tasks feed into the space-time predictor tasks or follow them, i.e. pass their result into predictor tasks of the subsequent time step. If errors affect these cheaper tasks that are not subject to our team checks, they will pollute follow-up space-time predictors to which our error detection and correction approach is applied again.

B. Asynchronous checking of task outcomes

The performance of our approach relies on a highly asynchronous implementation where teams are not running in lock step mode. They are not tightly synchronised. Therefore, a team is never allowed to block for receiving a redundantly computed result. Our implementation regularly checks for incoming task outcomes and receives them whenever available. Such a mechanism handles "unexpected" messages carrying task outcomes from tasks that have been executed earlier on the replicated rank than on the local rank. Polling the MPI subsystem prevents overflow of the MPI buffers in use and that MPI has to switch to a rendezvous protocol.

Once a task has been computed locally on a team A and its outcome is considered as dubious, there is no guarantee that team B delivers a matching outcome in a timely fashion (right branch of first check in Figure 1). Latency or contention delay any message delivery further. Busy waiting for the "control computation" therefore is not an option.

Whenever we wait for a replicated task's outcome, we want to switch to further computations while we wait. To be independent from modern MPI+X callback mechanisms [23] that support tasks with "interrupts" or listeners for incoming MPI messages, we introduce an additional task type which checks and corrects a computed dubious task result. This task is spawned for each dubious STP task that cannot be checked immediately. It is created with low-priority and reschedules itself until a redundantly computed result has been inserted into the local cache. Then, the consistency checks are performed and, if necessary, the task outcome is corrected. The tasks that reschedule themselves logically realise a polling mechanism, but the actual polling is spread out over further calculations as further tasks slot in.

C. Error model

Our experiments rely on a manual error injection to facilitate controlled studies. The underlying error model assumes that silent data corruption happens exclusively within the STP. It takes the STP's outcome $\hat{Q}$, and introduces errors by adding δQ to this outcome. We assume that no other errors arise, and that the silent data corruption exclusively affects the outcome of floating point calculations. This is reasonable, as data corruptions on integer data typically lead to wrong memory accesses or wrong execution logic, such that they materialise almost immediately in a hard error.

As our Q in (1) is represented by polynomials in a Lagrangian formalism over the cells, and as our error injection picks a sample point ξ_n and adds a value, a silent error alters the per-cell representation of a task outcome. In our task formalism, we obtain altered Lagrangian weights $\tilde{y}_i = y_i + e = \tau(\theta_i) + e$, which translates to a flawed space-time prediction $\hat{Q}(x,t) + \delta Q(x,t)$. An injected error thus alters the representation of the (predicted) solution in the entire cell, but does not immediately propagate globally.

Less than 20% of random bitflips within a given floating point number actually introduce non-negligible errors in $Q(x,t)$ for our application [9]. We therefore refrain from injecting an error into the task calculation or into θ_i. Instead, we fix a value $|e|$ and artificially add this value to one sample point ξ_n of one of the STP outcomes in one time step. This yields a guaranteed permutation δQ. We pick the error location randomly, i.e. a randomly chosen cell and a randomly chosen coefficient in its STP is affected. We also pick the affected time step randomly. There is only one single "bitflip" which manifests in a significant error, i.e. falls into the 20% category, and this "bitflip" is non-persistent, i.e. occurs only once.

V. Experimental results

We run all our tests on the SuperMUC-NG[1] supercomputer hosted by the Leibniz-Rechenzentrum in Garching. Each SuperMUC-NG compute node features two Intel Xeon Platinum 8174 CPUs ("Skylake" architecture) with 96 GBytes of main memory and 24 cores per CPU. Nodes are interconnected in a fat tree topology with Intel Omnipath. We compile and run ExaHyPE and teaMPI with the 2019 generation of the Intel compiler, Intel TBB and Intel MPI.

A. Sensitivity analysis

We first analyse sensitivity and correctnesss of the error criteria. We systematically prescribe different sizes of errors e (in contrast to Section IV-C discriminating between positive and negative errors), and then conduct 100 test runs for each fixed error size, randomly inserting that error once during the run. We track whether the injected error was detected and successfully corrected during the run, and thus compute the sensitivity rate, i.e. the total number of *corrected runs* divided by the total number of runs. In the case of a successfully corrected error, the solution remains unaffected by any corruption.

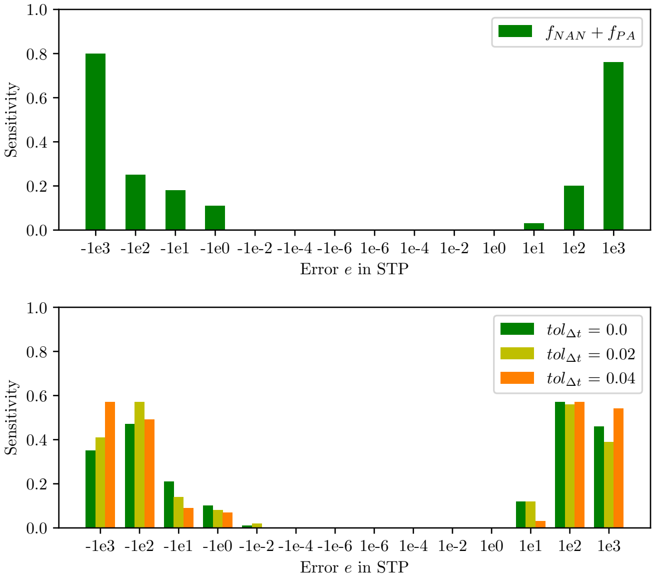

Fig. 3. Top: Error sensitivity for $f^{\mathrm{PA}} + f^{\mathrm{NaN}}$. Bottom: Error sensitivity for the time step sizes criterion $f_{\Delta t}$.

Injected but undetected errors or errors where the algorithm picks the wrong team as valid propagate and pollute the long-term wave field. We run all following parameter studies on a single node with two teams and with a single MPI rank per team. We report results only for numerical order 7 for brevity, although we obtained comparable results at other orders, too.

For the physical admissibility checks combined with the NaN search, the error size determines the sensitivity (Figure 3). The larger an error the more reliably it is detected and corrected. The NaN criterion is particularly robust, i.e. finds all NaN in the output (not shown), while the combined sensitivity is biased towards negative error contributions.

As our admissibility criterion searches for negative pressures or negative density values, its sensitivity is higher for negative error contributions compared to positive values. If the random error introduces a negative density in one sample point, it is clear that we have an error. However, also positive changes of any unknown can violate the admissibility: If the density is increased relative to the energy, the pressure reconstruction yields a negative pressure. The derived quantity harms the physical admissibility.

With the criterion $f_{\Delta t}$, our algorithm similarly reacts mostly to larger error magnitudes with sensitivity values of up to 50% (Figure 3). The sensitivity rate does not strongly correlate to the tolerance $tol_{\Delta t}$ and yields solely qualitative metrics, i.e. dubious vs. reasonable. Most smaller solution changes do not alter the time step size even if we compare the time step sizes bit-wisely, i.e. pick $tol_{\Delta t} = 0$. This property results from the fact that we use the maximum eigenvalue of the result to determine the admissible time step size. It is only a change that feeds into the maximum eigenvalue that also triggers the error criterion. While $f^{\mathrm{PAD}} + f^{\mathrm{NaN}}$ seems to yield stronger qualitative sensitivity statements for large errors, $f_{\Delta t}$ is more

[1] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG

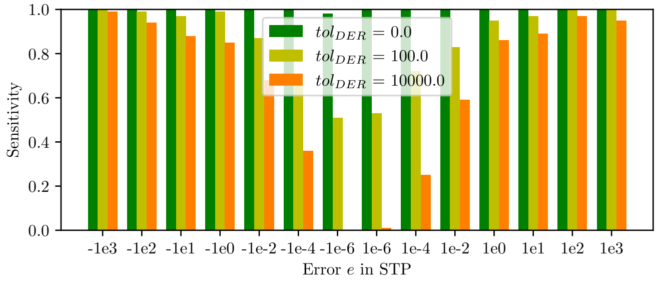

Fig. 4. Error sensitivity for the derivatives criterion f_{Der}.

TABLE I
AVERAGE ERROR SENSITIVITIES FOR DIFFERENT CONFIGURATIONS OF ERROR CRITERIA (ROUNDED TO TWO DECIMALS).

$tol_{\Delta t}$	tol_{Der}	$f_{\text{PA}} + f_{\text{NaN}}$	$f_{\Delta t}$	f_{Der}	all criteria (rig.)	all criteria (lazy)
0	0	0.17	0.16	1.00	1.00	1.00
0	100	0.17	0.16	0.86	1.00	0.83
0	10000	0.17	0.16	0.66	1.00	0.66
0.02	0	0.17	0.16	1.00	1.00	0.51
0.02	100	0.17	0.16	0.86	0.87	0.46
0.02	10000	0.17	0.16	0.66	0.77	0.37

sensitive for error in the order of $|e| \approx 10^2$.

The derivatives criterion is the most sensitive one (Figure 4): we can spot and correct errors with high sensitivity of > 0.8 in most cases. Sensitivity increases with lower tolerance tol_{Der} and—like the other criteria—with larger error magnitudes.

B. Combined dubiosity checks

We next investigate the combination of multiple criteria and compare averaged sensitivities for experiments using only one of the error criterion functions $f \in \{f^{\text{PA}}+f^{\text{NaN}}, f_{\Delta t}, f_{\text{der}}\}$ with experiments using either the rigorous or the lazy combination of *all* presented error criteria (Table I). In all cases but one, the sensitivity for the rigorous combination of all criteria is higher than the individual ones. The lazy combination yields a reduced sensitivity.

The lazy scheme skips some f_{Der} evaluations which mark a task outcome as dubious in the rigorous counterpart. It misses out on some dubious results. Yet, a combination of different criteria allows an additional calibration of the overall code's sensitivity beyond the tuning of tolerances, and thus yields more sensitive algorithmics for both the rigorous and lazy evaluation. While a maximum sensitivity of 1.0 can be obtained with the tolerances $tol_{\Delta t} = 0$ and $tol_{\text{Der}} = 0$, a rigorous combination of the error criteria comes at a performance price.

C. Performance

Both the choice of the error criterion functions as well as the respective thresholds are influential parameters as they determine how many task outcomes need to be validated, i.e. how many (additional) indicator evaluations we have to run (Figure 5). An optimal sensitivity of 1.0 (all error are recognised and fixed) can be obtained for many configurations where we evaluate f_{Der} always. However, the throughput is about a factor of three worse than the throughput of a code without any sensitivity check. This throughput statements refers to a single team run, i.e. we neglect savings due to the sharing of outcomes in a replicated world. On the other hand, the setup with the highest performance only exhibits an average sensitivity of ≈ 0.2. We observe a trade-off between performance and sensitivity. Configurations with lazy evaluation typically come at a penalty on sensitivity compared to their rigorous counterparts, but they achieve higher performance. Configurations with $tol_{\text{Der}} = 100$ (e.g., with lazy evaluation and $tol_{\Delta t} = 0$) give the best performance-accuracy trade-off: they achieve a sensitivity of around 0.8 while coming at a performance that is faster than fully redundant computation.

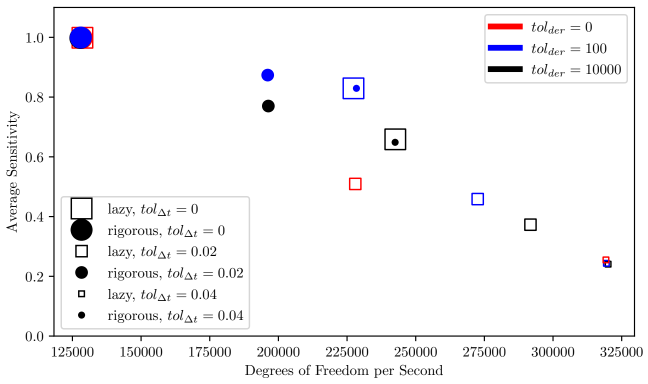

Fig. 5. Performance/sensitivity tradeoff of different configurations in a two rank setup (single injected error per run). The rigorous configurations with $tol_{\text{der}} = 0$ (red) are overlapped by others in the upper left corner.

D. Upscaling

We scale our benchmark on up to $35,088$ cores on SuperMUC-NG (Figure 6), where we compare the performance of different configurations. Each configuration runs two teams (each on up to $17,544$ cores). In the two baseline configurations, no errors are injected and our correction approach is disabled. Tasks are either computed redundantly (dashed black) or the two teams skip redundant computations using task outcome sharing [13] (dashed brown). Besides, there are three different variants with error injection and correction: (1) a rigorous variant with high sensitivity and high redundancy (red), (2) a lazy variant with lower sensitivity but less redundancy (green) and (3) a lazy variant which — in line with the results in Figure 5 — we may assume to have a good performance-accuracy trade-off (blue). In all runs, we measure the performance for 100 time steps. We inject (and correct) 10 errors in each run with error correction. The error values and spatial positions are hardcoded to obtain a controlled and deterministic setup. All experiments fix a number of MPI ranks for which our code results in a balanced domain decomposition of either a uniform grid with 25^3 cells (left half in Figure 6) or with 79^3 cells (right half in Figure 6). We then scale up the number of cores available to each rank.

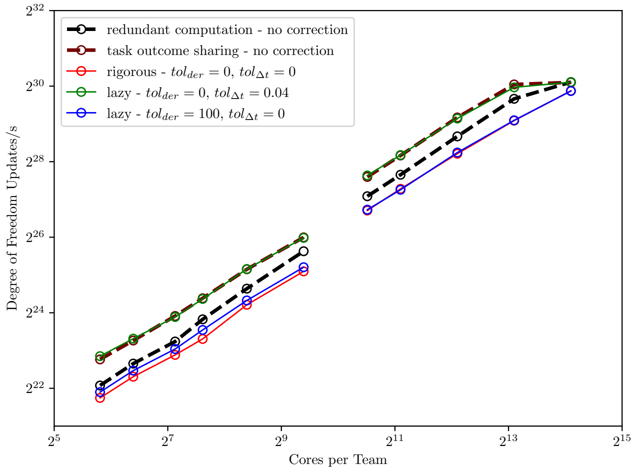

Fig. 6. Strong scaling of our benchmark on up to 35,088 cores. A grid of 25^3 cells is used for smaller core counts (left half, on 28 ranks), while a grid of 79^3 cells is employed for the larger setups (right half, on 731 ranks).

The rigorous variant, as well as the blue lazy variant, have a lower performance than the baseline with redundant computation. In both cases, error checking adds overhead for computing the error criteria, for transmitting outcomes and for waiting for their validation. Compared to the red rigorous variant, only a selected subset of all cells is checked in the blue lazy one, which explains why the blue variant performs better for the small grid. Yet, the non-uniform behavior of not validating all cells in the blue variant also creates load imbalances between ranks — a bottleneck, especially at high rank counts. The lazy green variant performs best, running at the full speed of the baseline that saves redundant computations, as error checking is applied to only few highly dubious cells. No error correction overhead is visible for this variant. All variants scale up similarly well, albeit small strong scaling effects at high core counts.

VI. Classification and Discussion

We next classify the properties of our approach, putting it into the context of related work where applicable.

a) Local vs. global analysis: Soft errors manifest in data corruption and can be found by running a full simulation at least twice and comparing both results [24]–[26]. While both runs may be executed in parallel, the comparison requires an offline or post-mortem analysis phase, once both redundant solutions are available. Differing solutions either imply silent corruption or an error in the application itself.

Such a *global* analysis challenges the supercomputer, as it introduces an explicit synchronisation between the two redundant runs, and as the subsequent comparison phase is very communication or I/O-heavy. Such bursts stress critical components of the machine. Instead, we adopt a *local* approach where task outcomes are compared while the computation is running. We thus avoid explicit synchronisation, and we spread out all comparisons over the whole simulation time.

b) Check granularity: Whenever we compare two redundantly computed solutions, equality is to be interpreted in a numerical sense. In a multi-threaded setting, the different orders of adding up individual partial numerical results may result in two byte-wise disagreeing solutions. A similar argument holds for complex cache access patterns where data is put from registers into memory and back.

The checks of redundantly computed results can happen at varying granularities: in principle, each individual floating point operation may be checked, i.e. we might compare data *bit-wisely* subject to floating point precision. In practice, this is often not feasible. On the other hand, we might operate with *global checksums* or hashes which map the whole solution onto one or few characteristic values and compare these. Our approach realises a bit-wise comparison in the tradition of the former approach. Yet, these checks are not automatically performed for all data all the time and we hence meet our non-functional requirement that the resiliency routines may not inacceptably increase the compute cost.

c) Redundancy level, check coverage and sensitivity: Redundant computations are the method of choice to spot errors and realise resiliency [12], [27]. While our approach technically offers *full redundancy*, i.e. two applications are in principle capable to run fully in parallel, the error criteria checks reduce the logical redundancy level: Sharing task outcomes implies that we logically work with a *partially redundant* setup where some data are assumed to be correct and are not cross-checked. There is no full coverage or validation of the outcomes.

We can only be *weakly confident* that we spot silent data corruptions. Strong errors resulting in NaNs or non-termination are always spotted by our algorithm. In return, our runtime data suggest that the sharing of confident outcomes allows us to reduce the overhead cost of the redundant calculation significantly. Resiliency is not for free, but it does not increase the compute cost by multiples of the baseline cost. As we cache redundant computations only temporarily, the permanent memory footprint also is not increased by multitudes.

Any error correction scheme must try to identify and correct potential soft errors as soon as possible to avoid error propagation. Ideally, an error is detected and corrected *immediately*, i.e., before it actually has the opportunity to affect any other numerical computations (immediate correction). As we abandon the idea of global bit-wise comparisons, we deploy the responsibility to identify errors immediately to the user code providing the error criterion functions.

d) Recovery strategies: Off-the-shelf solutions to recover from data corruption require simulations to run at least three times such that the code can rely on a *majority vote* to identify which results are valid. If one run is determined to be invalid, the whole simulation state is swapped, i.e. we continue after replicating a valid state. This is a *global recovery strategy*.

As we rely on cell-local error metrics, our approach realises a *local recovery strategy* where individual task outcomes are replaced if we spot an error. Furthermore, we rely solely on a *redundancy level of two*. Our error criteria replace the majority vote. This allows us to operate without an expensive synchronisation after a time step that brings a corrupted simulation back on track. We also do not have to store complete checkpoints or hold them in memory.

VII. Conclusion and outlook

We propose a task-based algorithmic framework which can recognise and correct soft errors. A clever combination of dubiosity metrics with task outcome sharing gives us an algorithm which exhibits almost the full robustness w.r.t. errors of a fully replicated run without the runtime penalty. Checkpointing is completely eliminated although we have to keep task outcomes in memory longer than the non-resilient baseline code and thus have a slightly increased memory footprint. While we have studied the impact for our ExaHyPE code and all of our work is open source,[2,3] the paradigms and ideas are of relevance for a large set of explicit time stepping codes and, to the best of our knowledge, the only known alternative to standard checkpoint-restart or full replication.

Natural follow-up work will combine the present ideas with algorithmic error correction techniques such as auto-correcting codes or limiter techniques. There are two further directions of future work worth exploring: On the one hand, a key ingredient of an efficient resilient realisation is the fast evaluation of the error criteria. The evaluation does not necessarily have to be done on a compute node. With smart network devices, task movement orchestration, labelling and merging may be offloaded into an intelligent network.

On the other hand, the choice of proper tolerances is worth investigating. In our experiments, we chose fixed particular tolerance combinations and highlighted how they affect the runtime and sensitivity. At the same time, our experiments "permitted" errors to happen only in one step of the overall computation. This fact plus the potential task divergence for dubious task outcomes imply that soft errors still can sneak into a computation and pollute the long-term behaviour. However, our data suggests that harsh sensitivity thresholds can find errors in any task—a propagated error can formally be seen as a newly added error in a cell—and recover from them. It is thus reasonable to experiment with dynamic thresholds which are typically rather relaxed. If a system suspects that errors start to creep in, it is reasonable to increase the sensitivity and thus to recover also from long-term errors.

References

[1] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild, "Applied mathematics research for exascale computing," Tech. Rep., 2014.

[2] M. Snir, R. Wisniewski, J. Abraham, S. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. Chien, P. Coteus, N. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, 2014.

[3] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. Dependable Secur. Comput.*, vol. 7, no. 4, 2010.

[4] G. Bronevetsky and B. R. de Supinski, *Soft error vulnerability of iterative linear algebra methods*, 2008.

[5] G. Bronevetsky, B. R. de Supinski, and M. Schulz, "A foundation for the accurate prediction of the soft error vulnerability of scientific applications," in *IEEE Workshop on Sil. Err. in Logic*, 2018.

[6] B. Austin, E. Roman, and X. Li, "Resilient matrix multiplication of hierarchical semi-separable matrices," in *FTXS*. ACM, 2015.

[7] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *ICS*. ACM, 2012.

[8] M. Altenbernd and D. Göddeke, "Soft fault detection and correction for multigrid," *Int. J. High Perf. Comp. Appl.*, vol. 32, no. 6, 2018.

[9] A. Reinarz, J.-M. Gallard, and M. Bader, "Influence of a-posteriori subcell limiting on fault frequency in higher-order dg schemes," in *FTXS*. IEEE TCHPC, 2018.

[10] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015.

[11] M. R. Varela, K. B. Ferreira, and R. Riesen, "Fault-tolerance for exascale systems," in *IEEE CLUSTER*, 2010.

[12] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC*. IEEE, 2012.

[13] P. Samfass, T. Weinzierl, B. Hazelwood, and M. Bader, "TeaMPI - replication-based resilience without the (performance) pain," in *ISC*, 2020.

[14] R. Melhem and T. Znati, "Lazy Shadowing - An adaptive, power-aware, resiliency framework for extreme scale computing," Tech. Rep., oct 2019.

[15] N. Gupta, J. R. Mayo, A. S. Lemoine, and H. Kaiser, "Towards distributed software resilience in asynchronous many- task programming models," in *FTXS*, 2020.

[16] A. Reinarz, D. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Koeppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl, "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems," *Comput. Phys. Commun.*, vol. 254, 2020.

[17] T. Weinzierl, "The Peano software - parallel, automaton-based, dynamically adaptive grid traversals," *ACM Trans. Math. Softw.*, vol. 45, no. 2, 2015.

[18] M. Dumbser and M. Käser, "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case," *Geophy. J. Int.*, vol. 167, no. 1, 2006.

[19] D. E. Charrier, B. Hazelwood, and T. Weinzierl, "Enclave tasking for DG methods on dynamically adaptive meshes," *SIAM J. Sci. Comput.*, vol. 42, no. 3, 2020.

[20] D. E. Charrier, B. Hazelwood, E. Tutlyaeva, M. Bader, M. Dumbser, A. Kudryavtsev, A. Moskovsky, and T. Weinzierl, "Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 5, 2019.

[21] L. Krenz, L. Rannabauer, and M. Bader, "A high-order discontinuous galerkin solver with dynamic adaptive mesh refinement to simulate cloud formation processes," in *PPAM*. Springer, 2019, pp. 311–323.

[22] J. F. Kelly and F. X. Giraldo, "Continuous and discontinuous Galerkin methods for a scalable three-dimensional nonhydrostatic atmospheric model: Limited-area mode," *J. Comput. Phys.*, vol. 231, no. 24, 2012.

[23] J. Schuchart, P. Samfass, C. Niethammer, J. Gracia, and G. Bosilca, "Callback-based completion notification using MPI continuations," *Parallel Comp.*, vol. 106, 2021.

[24] K. Mohanram and N. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *ITC*, 2003.

[25] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-directed instruction duplication for soft error detection," in *Design, Automation and Test in Europe*, 2005.

[26] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. Mannaru, A. Riska, and D. Milojicic, "Susceptibility of commodity systems and software to memory soft errors," *IEEE Trans. Computers*, vol. 53, no. 12, 2004.

[27] K. B. Ferreira, J. Stearley, J. H. L. III, R. Oldfield, K. T. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. C. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *SC*. ACM, 2011.

[2] https://gitlab.lrz.de/exahype/ExaHyPE-Engine

[3] https://gitlab.lrz.de/hpcsoftware/teaMPI