RESILIENCE FOR MASSIVELY PARALLEL MULTIGRID SOLVERS*
MARKUS HUBER†, BJÖRN GMEINER‡, ULRICH RÜDE‡, AND BARBARA WOHLMUTH†

Abstract. Fault tolerant massively parallel multigrid methods for elliptic partial differential equations are a step towards resilient solvers. Here, we combine domain partitioning with geometric multigrid methods to obtain fast and fault-robust solvers for three-dimensional problems. The recovery strategy is based on the redundant storage of ghost values, as they are commonly used in distributed memory parallel programs. In the case of a fault, the redundant interface values can be easily recovered, while the lost inner unknowns are recomputed approximately with recovery algorithms using multigrid cycles for solving a local Dirichlet problem. Different strategies are compared and evaluated with respect to performance, computational cost, and speedup. Especially effective are asynchronous strategies combining global solves with accelerated local recovery. By this, multiple faults can be fully compensated with respect to both the number of iterations and run-time. For illustration, we use a state-of-the-art petascale supercomputer to study failure scenarios when solving systems with up to $6 \cdot 10^{11}$ (0.6 trillion) unknowns.

Key words. fault tolerant algorithms, massively parallel and asynchronous multigrid

AMS subject classifications. 65N55, 65Y05, 68Q85

DOI. 10.1137/15M1026122

1. Introduction. Future high performance systems will be characterized by millions of compute nodes that may be executing up to a billion parallel threads. This computing power will be expensive not only with respect to acquisition cost but also due to the operational costs, whereby energy consumption is becoming a major concern. The increasing system size results in a higher probability of failure of one or more components of the high performance computing system [19, 30]. Recently, fault tolerance techniques and algorithms that support a fail-safe performance have become an active field of research.

Faults can be classified in fail-stop and fail-continue, also called hard and soft errors, respectively; see, e.g., [18, 20]. In the first case, the process stops, e.g., due to a permanent node crash or incorrect execution path that interrupts the program and results in a loss of the state of the process. In the case of soft errors, the process continues, but the failure affects the execution through “bit-flips.” Fault tolerance techniques can then be categorized as hardware-based fault tolerance (HBFT) [50, 53], system software–based fault tolerance (SBFT) [10, 12, 14, 29, 63], and algorithm-based fault tolerance (ABFT) [18, 22, 31, 43, 49]. For a general overview and a classification, we refer the reader to [19, 20, 21].

Achieving resilience is costly, since it always requires some form of redundancy, and thus extra system resources and energy. Traditional checkpoint strategies must

*Received by the editors June 15, 2015; accepted for publication (in revised form) August 5, 2016; published electronically October 27, 2016.

http://www.siam.org/journals/sisc/38-5/M102612.html

Funding: This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 “Software for Exascale Computing” (SPPEXA) and through WO671/11-1. The authors gratefully acknowledge the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN at Jülich Supercomputing Centre (JSC).

†Institute for Numerical Mathematics (M2), Technische Universität München, D-85748 Garching b. München, Germany (huber@ma.tum.de, wohlmuth@ma.tum.de).

‡Department of Computer Science 10, Friedrich-Alexander-Universität Erlangen-Nürnberg, D-91058 Erlangen, Germany (bjoern.gmeiner@fau.de, ulrich.ruede@fau.de).
collect and transfer the data regularly from all compute nodes and store the data to backup memory \[27, 44, 52, 57\]. In large systems, this may be quite expensive and slow. Consequently, alternatives exploiting specific features of the algorithms can be attractive. Under the assumption that the failure can be detected, such ABFT strategies achieve resilience in the algorithm exploiting application-specific properties. Originally, ABFT was proposed by Huang and Abraham \[43\] for systolic arrays where checksums monitor the data and are used for a reconstruction. Later, it was extended to applications in linear algebra, such as addition, matrix operations, scalar product, LU decomposition, and matrix transposition, and in fast Fourier transform methods \[4, 13, 24, 48\]. The work by Davies and Chen \[28\] deals with fault detection and correction during the calculation for dense matrix operations. For sparse matrix iterative solvers, such as SOR, GMRES, and CG iterations, the previously mentioned approaches are not suitable due to a high overhead \[58\] and were extended in \[18, 23, 56, 59\]. The influence of soft faults on an algebraic multigrid (AMG) solver is studied in \[22\]. A multigrid hierarchy is used in \[39\] to employ a data compression for check-pointing. Local recovery strategies for Krylov space solvers were studied in \[2, 47\]. In \[26\], Cui, Xu, and Zhang exploit the redundancy in the structure of a parallel subspace correction method.

At this time, the immediate detection and the replacement of the faulty components are not part of the standard message passing interface (MPI). However, fault tolerant MPI versions, such as Harness fault tolerant MPI (FT-MPI) \[32\] or user level failure migration (ULFM) \[10, 11\], are under development. Here, we consider fail-stop errors, as they may occur in iterative schemes when solving discretized elliptic partial differential equations (PDEs), and assume that faulty components are immediately detected. A new fault tolerant parallel geometric multigrid method is presented. Multigrid methods have the potential for excellent parallel efficiency and short execution times \[1, 5, 16, 25, 37, 38, 40, 45, 51, 54, 55, 60\]. We employ multigrid V-cycles where the coarsest grid grows with the processor number. As coarse grid solver, a modern direct sparse solver \[3\] could be used; see also \[55\]. Here we use an inexact preconditioned Krylov method as coarse grid solver. Although this is not asymptotically optimal in the processor number, we still achieve run-times of less than a minute for solving systems with up to \(6 \cdot 10^{11}\) unknowns on a quarter of the supercomputer JUQUEEN.

Similar to \[26\], we pursue fault tolerance strategies for hard faults that

- converge when a fault occurs, assuming it is detectable,
- minimize the delay in the solution process,
- minimize computational and communication overhead.

To compensate for a hard fault, we decouple the faulty part from the intact part temporarily and exploit the fact that only relatively little of the data must be stored redundantly so that a special recovery process can reconstruct the bulk of the missing data efficiently. Exploiting the flexibility of modern heterogeneous architectures, the fault can be compensated for if the recovery of the lost state is executed by a \textit{computational superman}. The data structures required for such a recovery can be found in many distributed memory parallel solvers that use domain partitioning and ghost nodes for communication.

The rest of our paper is organized as follows: In section 2, we describe the model PDE and fault setting, and in section 3, we discuss the data structure for our recovery strategies. The redundancy in the ghost layer data structures allows us to combine multigrid efficiency with tearing and interconnecting strategies. In section 4, we introduce local recovery strategies and study numerically their influence on the
global convergence. The main algorithmic result can be found in section 5, where Dirichlet–Neumann and Dirichlet–Dirichlet coupling strategies are combined with fast hierarchical multigrid methods [15]. In section 6, we test both new global recovery strategies on a petascale system. To compensate for the fault with respect to the time-to-solution, we enhance the massively parallel geometric multigrid method by asynchronous components on partitioned domains and study the single- and multiple-fault cases.

2. Model problem and fault setting. In this section, we introduce the model PDE scenario, the notation for the geometric multigrid solver, and the fault model for which we design our recovery techniques.

2.1. Model problem and parallel multigrid solver. For the sake of simplicity, we will illustrate the methods for the Laplace equation in three dimensions with inhomogeneous Dirichlet boundary conditions

\[
-\Delta u = 0 \quad \text{in } \Omega, \quad u = g \quad \text{on } \partial\Omega
\]

as the PDE model problem. The approach can be generalized to other types of boundary conditions, a nonhomogeneous right-hand side, or other PDE problems. In section 4, we will also present a numerical example for the Stokes system. We consider a bounded polyhedral domain \( \Omega \subset \mathbb{R}^3 \) that is triangulated with an unstructured base mesh \( \mathcal{T}_{-2} \). In the following, we will assume, for simplicity, that tetrahedral elements are used. From this initial coarse mesh, a hierarchy of meshes \( \mathcal{T} := \{T_l, l = 0, \ldots, L\} \) is constructed by successive uniform refinement; see, e.g., [9]. This hierarchy is used to construct the geometric multigrid solver.

The discretization of (2.1) uses conforming linear finite elements (FEs) on \( \mathcal{T} \) and leads canonically to a nested sequence of FE spaces \( V_0 \subset V_1 \subset \cdots \subset V_L \subset H^1(\Omega) \) and a corresponding family of linear systems

\[
A_l u_l = f_l, \quad l = 0, \ldots, L,
\]

where the Dirichlet boundary conditions are included. We apply multigrid correction schemes in V-cycles with standard components to (2.2). In particular, we use linear interpolation and its adjoint operator for the intergrid transfer, a hybrid variant of a Gauss–Seidel updating scheme with three pre- and postsmoothing steps [35, 42, 61] and a preconditioned conjugate gradient (PCG) method as coarse grid solver. All multigrid components can be replaced, e.g., an AMG method can be used as coarse grid solver, and do not affect the recovery algorithms directly. For pointwise smoothers in three dimensions, V(3,3)-cycles are found to be efficient with respect to residual and error reduction [38, 34, 62]. For a general overview of multigrid methods, we refer the reader to [16, 40].

In highly parallel multigrid frameworks for distributed memory architectures [1, 7, 33, 37, 60], the work load distribution for solving PDE problems is typically based on a geometrical domain partitioning. The base mesh \( \mathcal{T}_{-2} \) defines the partitioning used for parallelization. In our simple model case, all tetrahedra are equally refined and thus induce the same load, so that for the sake of simplicity, we assume that the number of processors is equal to the number of tetrahedra in \( \mathcal{T}_{-2} \). We thus have a one-to-one mapping also inducing an assignment for all refined meshes. In general, several of the coarse mesh elements would be assigned to each processor such that a good load balancing would be guaranteed.

Remark 1. We note that all of our techniques generalize to hexahedral and hybrid meshes as well as to higher-order or discontinuous Galerkin FEs.
2.2. Single-fault model and pollution effect. For our study, we concentrate on a specific fault model under assumptions as in [26, 41] and consider, for the moment, that only one faulty processing unit fails. A faulty processing unit can be, e.g., a processor or a compute node. Our methods deal with reconstructing the partly lost internal solver status. We point out that this is only one building block for achieving fault tolerance when, in general, the static process data also must be reconstructed.

The faulty subdomain $\Omega_F \subset \Omega$ is assumed to be a single open element in $\mathcal{T}_{-2}$, and $\Omega_I := \Omega \setminus \Omega_F$ is called an intact or healthy subdomain. Then, the intact and faulty regions are separated by an interface $\Gamma := \partial \Omega_I \cap \partial \Omega_F$. We denote the unknowns $u_F$, $u_I$, and $u_\Gamma$ with respect to the subdomains $\Omega_F$, $\Omega_I$ and the interface $\Gamma$, respectively; cf. Figure 1.

In case of a failure, the solution values in the faulty subdomain are set to zero as initial guess. After the local reinitialization of the problem and a possible recovery step, we continue with multigrid cycles in the solution process. We focus on three categories of job executions: A job execution in which no fault occurs is called a fault-free execution, a do-nothing execution is a job execution in which a failure occurs, but no special recovery strategy is applied (besides reinitializing the solution locally to zero), and a recovery execution stands for a job execution where recovery strategies are applied after a fault happens. A fault occurs after $k_F$ cycles, and we count the number of iterations necessary to reach the stopping criterion in the case of a fault-free execution $k_{\text{free}}$ and a faulty execution with possible recovery $k_{\text{faulty}}$. Similar to [30], we quantify the extra iterations compared to a fault-free execution. To do so, we introduce a relative cycle advantage parameter $\kappa$ defined by

$$\kappa := \frac{k_{\text{faulty}} - k_{\text{free}}}{k_F}.$$  

Intuitively, we expect that $\kappa \in [0, 1]$. The situation $\kappa = 0$ represents the case when the recovery algorithm reaches the stopping criterion with the same number of iterations as in a fault-free execution. For increasing $\kappa$, more and more additional iterations are required to reach the stopping criterion. For $\kappa = 1$, $k_F$ additional multigrid iterations have to be carried out, and this means that essentially all information that had been accumulated before the fault has been lost.

The following simple test setting illustrates the need for special recovery strategies. We consider a scenario with 16 million unknowns in the unit cube $\Omega = (0,1)^3$ and a loss of 0.3 million unknowns due to a fault. The faulty subdomain is located as shown in Figure 1. On the left of Figure 2, we illustrate the reduction of the residual within the specified V-cycle multigrid scheme. The fault occurs after $k_F = 5$ multigrid iterations, and in the case of a do-nothing execution, the residual after the fault is highly increased, resulting in four additional multigrid steps to obtain a given
tolerance compared to the fault-free execution. Further tests show that the number of additional steps is almost always equal to $k_F - 1$. Thus, a do-nothing execution results in a $\kappa$ close to one. A favorable preasymptotic convergence rate after the fault often helps to save one cycle, but aside from this, the extra cost incurred by the fault is essentially the number of cycles that have been performed before the fault. The situation becomes even worse when multiple faults occur.

![Image of residual decay](image1.png)

**Fig. 2.** Residual decay (left) and residual distribution after the failure (middle) and after one additional global V-cycle (right) restricted to a cross section through the domain $\Omega$ ($\alpha := \log_{10}(\text{Residual})$, where Residual stands for a properly scaled Euclidean norm of the residual). Note the different scales in the middle and the right figure.

The middle illustration in Figure 2 visualizes the residual on a cross section through the domain directly after the failure and reinitialization. The residual distribution when one additional global multigrid cycle has been performed is given on the right. Note that large local error components within $\Omega_F$ are dispersed globally by the multigrid cycle. Although the smoother transports information only across a few neighboring mesh cells on each level, their combined contributions on all grid levels lead to a global pollution of the error. While the residual decreases globally, the residual in the healthy domain $\Omega_I$ increases by this pollution effect. A possible remedy is based on temporarily decoupling the domains to prevent the locally large residuals from polluting the healthy domain.

3. Abstract framework for algorithmic resilience. In this section, we discuss a software architecture that is suitable to dealing with faults and that supports numerical recovery strategies. The main ingredient for the recovery algorithms is a hybrid data structure that allows us to combine multigrid mesh hierarchies with tearing and interconnecting strategies from domain partitioning. All of our numerical results will be carried out within the hierarchical hybrid grids (HHG) software framework [8, 36, 37], but the essential techniques can be adapted to other implementations, since they only exploit the redundancy of data in ghost layers, as in other distributed memory parallel frameworks.

3.1. Hybrid grid data structures. In parallel, the communication is often realized by ghost layers (sometimes also called halos) which redundantly store copies of master data. This is a convenient technique for accommodating data dependencies across boundaries. Only the original master values can be written by the algorithm, and thus, once a master value has been changed, the associated ghost values must be updated to hold consistent values. Here, we briefly discuss a systematic construction of the ghost layer data structures that is induced by the mesh geometry.

Note that the refined meshes will have nodes that are located on the vertices, edges, faces, and the interior of the initial elements in $T_{-2}$. For two tetrahedra in
three dimensions, this is illustrated in Figure 3. This classification of nodes induces a system of container data structures. In particular, there are three-dimensional (3D) containers to hold the nodes in the interior of each \( T_i \in \mathcal{T}_2 \), then 2D containers for those nodes that lie on each coarse mesh face \( F_{i,j} = \partial T_i \cap \partial T_j \), 1D containers for the nodes that lie on the coarse mesh edges, and eventually 0D containers that hold trivially the nodes coinciding with the vertices of \( \mathcal{T}_2 \).

Conceptually, the next step is to introduce the ghost layers. The geometrical classification above induces a distribution of the master copy of each node into a unique container. These containers can be enriched by ghost nodes which are copies of master nodes that are stored elsewhere. Thus all the fine grid nodes that rest on the boundary of a \( T_i \in \mathcal{T}_2 \) become ghost nodes in that \( T_i \); similarly, in the face data structure, the nodes that lie on the edges become ghost nodes; and eventually the end points of edges become ghost nodes for the edge data container. Furthermore, each of the face, edge, and node containers is enriched by one additional layer of ghost nodes that hold additional copies of master values. These extra ghost layers are essential for, e.g., efficiently implementing local relaxation algorithms for the master nodes of the corresponding face container. For the parallelization, we eventually introduce additional copies of the face, edge, and vertex containers so that they can be allocated uniquely on the processors of a distributed memory system. In the HHG implementation, the communication algorithm between nodes is thus split into two steps. The first consists in updating the ghost node values in the containers that are local to a processor by simple local copy operations. The second step is the actual communication of the interface data structures (face, edge, node) and synchronization by message passing in the distributed memory system.

We remark that this construction obviously leads to redundancy in face, edge, and node data and to extra cost in data storage, but that the additional memory cost is of lower-order complexity. As we will see below, this redundancy enables efficient implementations of the parallel multilevel solver algorithms, including a systematic recovery after a fault, i.e., the numerical recovery of the data in these structures. If a component failure leads to the loss of a master face (or edge or node), then the recovery can be directly performed using the redundant information. Thus, in the following we focus only on the algorithmic recovery of data in 3D containers. Note that the interface data structures can also be used to implement Dirichlet or Neumann boundary conditions for those elements that lie at the domain boundary.

### 3.2. Equivalent algebraic formulations

The data structure discussed in the previous subsection allows different equivalent algebraic formulations of (2.2) that play an important role in the following sections.
Figure 3 illustrates the ghost layer structure of a coarse mesh face with the master and ghost nodes of a refined mesh. For the interface container structures (faces, edges, vertices), a recovery is directly possible, since we can assume that redundant copies exist in the system. However, for the volume elements, their inner node values are not available, and thus they must be reconstructed numerically. The ghost layer structure and the associated data redundancy can be represented algebraically by rewriting (2.2) equivalently as

\[
\begin{pmatrix}
A_{II} & A_{II} & 0 & 0 & 0 & \\
0 & Id & -Id & 0 & 0 & \\
A_{II} & 0 & A_{II} & 0 & A_{II} & \\
0 & 0 & -Id & Id & 0 & \\
0 & 0 & 0 & A_{II} & A_{II} & \\
\end{pmatrix}
\begin{pmatrix}
u_I \\
u_{\Gamma_I} \\
u_{\Gamma_F} \\
u_{\Gamma_F} \\
u_{\Gamma_F} \\
u_{\Gamma_F} \\
\end{pmatrix} =
\begin{pmatrix}
f_I \\
f_{\Gamma_I} \\
f_{\Gamma_F} \\
f_{\Gamma_F} \\
f_{\Gamma_F} \\
f_{\Gamma_F} \\
\end{pmatrix}.
\]

The submatrices are associated with the block unknowns; in more general settings, they also depend on the basis functions and the PDE. Because of the locality of the support of the basis function, we can identify \(A_{\Gamma_I}\) with \(A_{\Gamma_I}^{\top}\) and \(A_{\Gamma_F}\) with \(A_{\Gamma_F}^{\top}\). Rows 2 and 4 in (3.1) guarantee the consistency of the redundant data at the interface between the master elements. Row 3 reflects the ghost layer structures associated with the master faces. We recall that by our assumptions the data \(u_{\Gamma_I}\) and \(u_{\Gamma_F}\) are lost, but \(u_I\) and \(u_{\Gamma_I}\) are still available. Although it is a priori not known what the intact domain will be, the data structure always allows such a presentation.

The ghost layer structure not only permits access to \(A_{\Gamma_I}\), but also decomposes it into \(A_{\Gamma_I} + A_{\Gamma_F}\). More precisely, for each vertex on a master face, the two substencils associated with \(\Omega_I\) and \(\Omega_F\) are available; see also Figure 4. Having the substencils at hand, we can rewrite (2.2) equivalently in terms of the additional flux unknown \(\Delta_{\Gamma_I}\) as

\[
\begin{pmatrix}
A_{II} & A_{II} & 0 & 0 & 0 & 0 & \\
A_{II} & Id & 0 & -Id & 0 & 0 & \\
A_{II} & 0 & Id & 0 & -Id & 0 & \\
0 & 0 & 0 & A_{II} & A_{II} & A_{II} & \\
0 & 0 & 0 & 0 & A_{II} & A_{II} & A_{II} & \\
\end{pmatrix}
\begin{pmatrix}
u_I \\
u_{\Gamma_I} \\
u_{\Gamma_F} \\
u_{\Gamma_F} \\
u_{\Gamma_F} \\
u_{\Gamma_F} \\
\end{pmatrix} =
\begin{pmatrix}
f_I \\
f_{\Gamma_I} \\
f_{\Gamma_F} \\
f_{\Gamma_F} \\
f_{\Gamma_F} \\
f_{\Gamma_F} \\
\end{pmatrix}.
\]

Here, \(-\Delta_{\Gamma_I}\) stands for the discrete flux out of \(\Omega_I\) and into \(\Omega_F\). It reflects a Neumann boundary condition for the healthy domain. In (3.1) and (3.2), we thus find the typical algebraic structure of Dirichlet and Neumann subproblems associated with the faulty and the healthy subdomain, respectively. This observation motivates the design of our parallel recovery algorithms.

4. Local recovery strategy. To avoid the error pollution effect observed in subsection 2.2, we now propose a first recovery strategy. We separate the healthy from the faulty subdomain, that is, we set the faulty subdomain in quarantine until the lost values in \(u_F\) can be recovered by solving a local auxiliary discrete Dirichlet problem with boundary data \(u_{\Gamma_I}\) on \(\Gamma\). Before continuing with the global multigrid iteration for solving (2.2), we approximate the auxiliary local problem by \(n_F\) steps of a given iterative solver; see Algorithm 1. Obviously, this algorithm is only a first step towards a fault tolerant multigrid solver and suffers from the fact that during the approximation of the local problem on \(\Omega_F\), the processors assigned to the healthy domain \(\Omega_I\) will be idle. Since in practice \(\Omega_F\) is small compared to \(\Omega_I\), this will
naturally result in a performance loss. A remedy will be addressed in section 5. Nevertheless, the local strategies will already provide valuable insight in the selection of the iterative solver for the recovery subproblem and in the choice of $n_F$.

Algorithm 1. Local recovery (LR) algorithm.

1. Solve (2.2) by multigrid cycles.
2. if Fault has occurred then
   3. STOP solving (2.2).
   4. Recover the boundary data $\overline{u}_{F}^\Gamma$ from line 4 in (3.1).
   5. Initialize $\overline{u}_{F}$ with zero.
   6. Approximate line 5 in (3.1) by solving iteratively with $n_F$ steps:
      7. $A_{FF} \overline{u}_F = f_F - A_{F \Gamma} \overline{u}_{\Gamma}^\Gamma$.
   8. RETURN to line 1 with new values $\overline{u}_F$ in $\Omega$.
   9. end if

4.1. Influence of the algorithmic parameters on the performance. To illustrate the method, we consider again the situation of subsection 2.2. We study the influence of the algorithmic parameters by setting an extremely small stopping criterion ($10^{-14}$) for the relative residual in a properly scaled Euclidean norm. In section 6, we also consider a level-dependent stopping criterion.

We study three different iterative solvers: local V-cycles and local W-cycles, each with three pre- and postsmoothing steps of the hybrid Gauss–Seidel smoother, and Gauss–Seidel relaxation only (Smooth); see the left picture in Figure 5.

The different local multigrid cycles, i.e., V- and W-cycles, perform similarly well. In these cases, setting $n_F = 2$ results in a cycle advantage of $\kappa = 1/5$. For Smooth, we report on two choices for $n_F$. The first one with $n_F = 15$ results in a computational cost similar to that of the V-cycle with $n_F = 2$, but no improvement compared to the do-nothing execution can be observed. The second one achieves $\kappa = 1/5$ but needs $n_F = 2000$, an absurdly high computational cost. Thus, as expected, Smooth is not a suitable candidate for solving the local recovery (LR) subproblem. On the right of Figure 5, we show the influence of $n_F$ for the case of a local V-cycle multigrid solver. The delay in finding the solution significantly depends on how accurately the LR problem is solved. Here, performing one local V-cycle for recovery reduces the iterations in the subsequent global solution process by two iterations. Two local V-cycles produce an advantage of three global iterations, and three local V-cycles completely compensate for the effect of the fault.

Table 1 shows the influence of the iteration type, $k_F$, and $n_F$ on $\kappa$ in more detail. We consider three cases for $k_F$, i.e., $k_F \in \{5, 7, 11\}$, the three iterative solvers from...
4.2. Extension to Stokes. The LR strategy is not limited to positive definite systems but can also be applied to saddle-point problems. As a model problem for above, and $n_F$ ranging from 1 to 6500. We additionally include the Jacobi PCG iteration as local solver.

In all settings, we achieve $\kappa < 1$. However, as can be expected, there are huge differences in the performance. In the case of PCG and Smooth, the number of required extra multigrid steps is exactly the same as in a do-nothing execution if we set $n_F = 50$ and $n_F = 500$, respectively. Only by increasing $n_F$ significantly do we observe a decrease of $\kappa$. In the case of recovery by Gauss–Seidel smoothing iterations and $k_F = 5, 7, 11$, we can compensate for the fault with $n_F = 3000, 4000, 6500$, respectively, and for a PCG recovery $n_F = 200, 250, 350$ is required. Obviously, performing a recovery with Gauss–Seidel and PCG iterations does not result in a recovery strategy of any computational interest in large-scale simulations. Thus, only the case of multigrid cycles is of practical relevance. Here, due to a preasymptotic behavior after the failure, we already obtain $\kappa = 0$ with $n_F = k_F - 2$.

Table 1
Cycle advantage for an early (after 5 iterations), middle (after 7 iterations), and late (after 11 iterations) fault for different $n_F$ and iterative solvers.

<table>
<thead>
<tr>
<th>Fault after 5 iter.</th>
<th>Strategies $\kappa$</th>
<th>Fault after 7 iter.</th>
<th>Strategies $\kappa$</th>
<th>Fault after 11 iter.</th>
<th>Strategies $\kappa$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No rec.</td>
<td>0.80</td>
<td>No rec.</td>
<td>0.86</td>
<td>No rec.</td>
<td>0.91</td>
</tr>
<tr>
<td>1 × V-cycle</td>
<td>0.60</td>
<td>1 × V-cycle</td>
<td>0.71</td>
<td>1 × V-cycle</td>
<td>0.91</td>
</tr>
<tr>
<td>2 × V-cycle</td>
<td>0.20</td>
<td>3 × V-cycle</td>
<td>0.29</td>
<td>5 × V-cycle</td>
<td>0.36</td>
</tr>
<tr>
<td>3 × V-cycle</td>
<td>0.00</td>
<td>5 × V-cycle</td>
<td>0.00</td>
<td>9 × V-cycle</td>
<td>0.00</td>
</tr>
<tr>
<td>1 × W-cycle</td>
<td>0.80</td>
<td>1 × W-cycle</td>
<td>0.71</td>
<td>5 × W-cycle</td>
<td>0.36</td>
</tr>
<tr>
<td>2 × W-cycle</td>
<td>0.20</td>
<td>3 × W-cycle</td>
<td>0.29</td>
<td>9 × W-cycle</td>
<td>0.00</td>
</tr>
<tr>
<td>3 × W-cycle</td>
<td>0.00</td>
<td>5 × W-cycle</td>
<td>0.00</td>
<td>9 × W-cycle</td>
<td>0.00</td>
</tr>
<tr>
<td>50 × PCG</td>
<td>0.80</td>
<td>50 × PCG</td>
<td>0.86</td>
<td>50 × PCG</td>
<td>0.91</td>
</tr>
<tr>
<td>100 × PCG</td>
<td>0.60</td>
<td>150 × PCG</td>
<td>0.43</td>
<td>150 × PCG</td>
<td>0.64</td>
</tr>
<tr>
<td>200 × PCG</td>
<td>0.00</td>
<td>250 × PCG</td>
<td>0.00</td>
<td>350 × PCG</td>
<td>0.00</td>
</tr>
<tr>
<td>500 × Smooth</td>
<td>0.80</td>
<td>500 × Smooth</td>
<td>0.86</td>
<td>500 × Smooth</td>
<td>0.91</td>
</tr>
<tr>
<td>2000 × Smooth</td>
<td>0.20</td>
<td>2000 × Smooth</td>
<td>0.43</td>
<td>2000 × Smooth</td>
<td>0.64</td>
</tr>
<tr>
<td>3000 × Smooth</td>
<td>0.00</td>
<td>4000 × Smooth</td>
<td>0.00</td>
<td>6500 × Smooth</td>
<td>0.00</td>
</tr>
</tbody>
</table>
an indefinite linear equation system, we consider a stabilized discretization of the 
Stokes equations. We use linear equal-order FE spaces and ensure stability by a 
standard pressure stabilized Petrov–Galerkin (PSPG) method; see, e.g., [17]. To 
solve the system efficiently, we apply a geometric multigrid method in the pressure 
and velocity with the same standard intergrid transfer components as before. The 
smoother reflects the indefinite structure and is of Uzawa type; see, e.g., [6, 64].

In the case of a failure, we recompute the values of the velocity and pressure 
by local solves of an auxiliary Stokes problem with Dirichlet boundary condition. In 
contrast to the standard situation, we also fix the pressure at the boundary $\Gamma$. To 
guarantee that the system is uniformly well-posed, we formally include a compatibility 
condition obtained from the normal components of the velocity. Then, the structure 
of Algorithm 1 can be directly used.

In Figure 6, a test case is set up in which a fault occurs after 5 iterations. We 
apply different numbers of local V-cycles to recover the lost values. Similar to the 
positive definite case in subsection 4.1, an LR by $n_F = k_F - 2$ results in $\kappa = 0$.

![Convergence of the relative residual of the Stokes system for different $n_F$.](image)

**Fig. 6.** Convergence of the relative residual of the Stokes system for different $n_F$. 

5. **Global recovery strategy.** So far we have only considered LR strategies in 
the faulty subdomain. During the recovery, the processors in the healthy subdomain 
are halted until the recovery in the faulty subdomain has terminated. However, for the 
overall performance, we cannot neglect the time spent in the recovery process. Thus, 
we extend our approach in two directions. First, we introduce a superman strategy 
to speed up the LR process itself, and second, we propose a global recovery such that 
both the faulty and the healthy subdomain, a subtask is executed asynchronously 
in parallel.

5.1. **The local superman.** In parallel geometric multigrid, load balancing often 
can be based on the number of degrees of freedom in subdomains. Here, the situation 
is quite different. The multigrid iteration in the faulty subdomain must deal with a 
much higher residual compared to the residual on the healthy domain. To compensate 
for this, we use local superman units, i.e., we overbalance the computing power with 
respect to the number of degrees of freedom in the faulty subdomain. Technically, 
the additional computing resources for realizing the superman are of the size of the 
faulty processing unit and can be provided by additional parallelization, under 
the assumption that using more processors for the faulty subdomain leads to a significant 
speedup. We propose here that, e.g., one or more full (shared memory) compute 
nodes are assigned to perform the LR for a domain that was previously handled by a
single processor or compute node, respectively. In our case, this can be accomplished by a tailored OpenMP parallelization for the recovery process in the shared memory case. Alternatively, a further domain partitioning of $\Omega_F$ can be used together with an additional local distributed memory parallelization. One may also think to exploit accelerators, such as graphics processing units (GPUs) or a Xeon Phi.

We denote by $\eta_s$ the ratio of the computing power on $\Omega_F$ after and before the fault. In addition to $n_F$, we introduce $n_I$ as the number of iterative solver steps on $\Omega_I$ which are carried out in parallel to the $n_F$ local solver steps on $\Omega_F$. Since local and global computation are performed simultaneously, the time for the recovery is presented as $\max(t_I, t_F)$, where $t_k$, $k \in \{F, I\}$, is the time spent on $\Omega_k$. For the moment, we fix $\eta_s \in \mathbb{N}$ small and set $n_F = \eta_s n_I$. This choice is motivated by the assumption that the time $\Delta t_F$ per local solve on $\Omega_F$ is bounded by $\Delta t_I / \eta_s$, where $\Delta t_I$ is the time per solve on $\Omega_I$. This assumption can be either too optimistic, e.g., if the local solves already operate near the strong scaling limit, or too pessimistic, e.g., if the coarse grid solver contributes more significantly to the run-time on $\Omega_I$ than on $\Omega_F$. We will account for these effects in section 6, where $n_F$ is set dynamically and leads to an asynchronous recovery algorithm.

For the ideal but hypothetical situation $\eta_s = \infty$, the LR of section 4 does not contribute to the global run-time. In section 6, we will set $n_F$ dynamically within the algorithm. By doing so, we can benefit from smaller relative local run-times. Typically on $\Omega_F$, the number of unknowns on the coarse mesh is quite small, and in comparison to $\Omega_I$, message passing communication is reduced. Here, we present the basic form of our recovery algorithms. We have two different types depending on the selected boundary condition for $\Omega_I$. The overhead with respect to the number of processes introduced by reserving the superman structure is almost negligible in large-scale computations. In our largest experiments in section 6.1, we perform simulations with 215,622 MPI processes and reserve only up to 8 spare MPI processes for the recovery. This constitutes an overhead of less than 0.004%. However, the communication organization in networks with reserved processes can have a significant impact on the performance. Constructing networks to mitigate this impact is a nontrivial design consideration and a future research area.

5.2. Dirichlet–Dirichlet recovery strategy. In the Dirichlet–Dirichlet recovery, we freeze the values $u_{\Gamma_I}$ at the interface $\Gamma_I$. This allows us to compute the two subproblems in $\Omega_F$ and $\Omega_I$ independently, and consequently, no communication between $\Omega_F$ and $\Omega_I$ is necessary. Thus, it is guaranteed that no defect data pollute the healthy subdomain during the recovery. On both subdomains, we iterate now on decoupled Dirichlet problems with boundary data on $\Gamma$ given by $u_{\Gamma}$. Obviously, at some point we have to reconnect the two subdomains again. The algorithm for the Dirichlet–Dirichlet recovery is presented in Algorithm 2. Additional communication (collective) is required in line 3 to inform all processors about the fault, and in line 4 (point-to-point) all units holding interface data have to send them to the replacements. In line 11, we reconnect both domains (collective) and synchronize the interface (point-to-point).

5.3. Dirichlet–Neumann recovery strategy. In the Dirichlet–Neumann recovery strategy, we do not freeze the interface values but treat them as Neumann boundary data in the healthy subdomain. By doing so, we use a one-directional coupling between the faulty subdomain $\Omega_F$ and the healthy subdomain $\Omega_I$. On $\Omega_I$, we approximate a Neumann boundary problem with static data, whereas on $\Omega_F$, we approximate a Dirichlet problem with dynamic boundary data. After each multigrid cycle on $\Omega_I$, the newly computed interface values $u_{\Gamma_I}$ are communicated via $u_{\Gamma}$ onto

1: Solve (2.2) by multigrid cycles.
2: if Fault has occurred then
3: STOP solving (2.2).
4: Recover boundary data \( u_{F_F} \) from line 4 in (3.1).
5: Initialize \( u_F \) with zero.
6: In parallel do:
7: (a) Use \( n_F \) multigrid cycles accelerated by \( \eta_s \) to approximate line 5 in (3.1):
8: \[ A_{FF} u_F = f_F - A_{F_F} u_{F_F}. \]
9: (b) Use \( n_I \) multigrid cycles to approximate line 1 in (3.1):
10: \[ A_{II} u_I = f_I - A_{I_I} u_{I_I}. \]
11: RETURN to line 1 with new values \( u_I \) in \( \Omega_I \) and \( u_F \) in \( \Omega_F \).
12: end if

\( u_{F_F} \). Hence, we only avoid communication from the faulty to the healthy domain but still keep communicating from the intact to the faulty subdomain. As in the case of the Dirichlet–Dirichlet recovery strategy, it is necessary to fully interconnect both subdomains after several cycles. The algorithm is presented in Algorithm 3. Again, additional communication (collective) is necessary in line 3. Further, we need to communicate the interface values \( u_I \) in line 6 and line 15 (point-to-point), respectively.


1: Solve (2.2) by multigrid cycles.
2: if Fault has occurred then
3: STOP solving (2.2).
4: Compute the Neumann condition from line 2 in (3.2):
5: \[ \tilde{\lambda}_I = -A_{II} u_I - A_{I_I} u_{I_I}. \]
6: Recover boundary data \( u_{F_F} \) from line 5 in (3.2).
7: Initialize \( u_F \) with zero.
8: for iter = 1, ..., \( n_I \) do
9: In parallel do:
10: (a) Perform \( n_F/n_I \) multigrid cycles to approximate line 6 in (3.2) accelerated by \( \eta_s \):
11: \[ A_{FF} u_F = f_F - A_{F_F} u_{F_F}. \]
12: (b) Perform 1 multigrid cycle to approximate lines 1 and 2 in (3.2):
13: \[ \begin{pmatrix} A_{II} & A_{I_I} \\ A_{I_I} & A_{I_I} \end{pmatrix} \begin{pmatrix} u_I \\ u_{I_I} \end{pmatrix} = \begin{pmatrix} f_I \\ A_{I_I} \end{pmatrix}. \]
14: Update boundary values \( u_I \) from line 3 in (3.2).
15: end for
16: RETURN to line 1 with values \( u_I \) in \( \Omega_I \), \( u_I \) on \( \Gamma \), and \( u_F \) in \( \Omega_F \).
17: end if

5.4. Comparison of the recovery strategies. In this subsection, we compare the Dirichlet–Dirichlet (DD) and the Dirichlet–Neumann (DN) recovery with the LR strategy, where calculations are performed only in the faulty subdomain, while the healthy domain stays idle.

From left to right in Figure 7 are shown the three different fault geometries and macromeshes, which are uniformly refined seven times for the following computations.
Scenario I has 2 million unknowns, and the fault (red-colored tetrahedron; color available online) is located with two faces at the boundary of the computational domain and affects about 16.7% of the total unknowns. Scenario II has 16 million unknowns, a corruption by the fault of 2.0% (as in subsections 4.1 and 2.2), and the faulty domain has one edge and three nodes coinciding with the boundary. Scenario III has 56 million unknowns, and the faulty domain is floating in the interior of Ω and affects only 0.6% of the unknowns.

Fig. 7. From left to right: Scenario I, Scenario II, and Scenario III.

In Tables 2 and 3, we present the cycle advantage values for $k_F = 5$ and $k_F = 11$, respectively, computed by formula (2.3). We compare our strategies for two different superman speed-up factors, $\eta_s = 2$ and $\eta_s = 5$. These superman factors are for the moment arbitrary and represent a small but nontrivial speedup and a higher speedup. In addition, we also illustrate the influence of $n_I$ in the recovery strategy before interconnecting the faulty and healthy domains. Note that the choice $n_I = 0 = n_F$ reflects a do-nothing execution. The numerical tests for all cases are run for $n_I \in \{0, 1, 2, 3, 4\}$, and for the case $k_F = 11$ and $\eta_s = 2$, they are also run for $n_I \in \{5, 6\}$. To evaluate $\kappa$ given by (2.3), we set $k_{\text{faulty}}$ to the sum of the required multigrid cycles on $\Omega$ and $n_F/\eta_s$. We note that the LR of section 4 can be also regarded as a special case of Algorithm 2, where formally $n_I$ is always equal to zero. For the DD and DN strategies, we have $n_I = n_F/\eta_s$.

### Table 2
Comparison of the cycle advantage $\kappa$ for the local, DD, and DN recovery strategies for a fault after 5 iterations.

<table>
<thead>
<tr>
<th>$n_F$</th>
<th>LR</th>
<th>DD</th>
<th>DN</th>
<th>LR</th>
<th>DD</th>
<th>DN</th>
<th>LR</th>
<th>DD</th>
<th>DN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scenario I</td>
<td></td>
<td></td>
<td>Scenario II</td>
<td></td>
<td></td>
<td>Scenario III</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
</tr>
<tr>
<td>2</td>
<td>0.20</td>
<td>0.20</td>
<td>0.00</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0.40</td>
<td>0.20</td>
<td>0.00</td>
<td>0.20</td>
<td>0.20</td>
<td>0.00</td>
<td>0.40</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.60</td>
<td>0.40</td>
<td>0.40</td>
<td>0.40</td>
<td>0.40</td>
<td>0.20</td>
<td>0.60</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.80</td>
<td>0.60</td>
<td>0.60</td>
<td>0.60</td>
<td>0.60</td>
<td>0.40</td>
<td>0.80</td>
<td>0.40</td>
<td>0.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n_F$</th>
<th>LR</th>
<th>DD</th>
<th>DN</th>
<th>LR</th>
<th>DD</th>
<th>DN</th>
<th>LR</th>
<th>DD</th>
<th>DN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scenario I</td>
<td></td>
<td></td>
<td>Scenario II</td>
<td></td>
<td></td>
<td>Scenario III</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
</tr>
<tr>
<td>5</td>
<td>0.20</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.20</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>0.40</td>
<td>0.20</td>
<td>0.00</td>
<td>0.20</td>
<td>0.20</td>
<td>0.00</td>
<td>0.40</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>15</td>
<td>0.60</td>
<td>0.40</td>
<td>0.20</td>
<td>0.40</td>
<td>0.40</td>
<td>0.20</td>
<td>0.60</td>
<td>0.20</td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>0.80</td>
<td>0.60</td>
<td>0.40</td>
<td>0.60</td>
<td>0.60</td>
<td>0.40</td>
<td>0.80</td>
<td>0.40</td>
<td>0.20</td>
</tr>
</tbody>
</table>
Table 3  
Comparison of the cycle advantage $\kappa$ for the local, DD, and DN recovery strategies for a fault after 11 iterations.

<table>
<thead>
<tr>
<th>$n_F$</th>
<th>Scenario I</th>
<th>Scenario II</th>
<th>Scenario III</th>
<th>Scenario III</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LR DD DN</td>
<td>LR DD DN</td>
<td>LR DD DN</td>
<td>LR DD DN</td>
</tr>
<tr>
<td>0</td>
<td>0.82 0.82 0.82</td>
<td>0.82 0.82 0.82</td>
<td>0.91 0.91 0.91</td>
<td>0.91 0.91 0.91</td>
</tr>
<tr>
<td>2</td>
<td>0.64 0.64 0.64</td>
<td>0.64 0.64 0.64</td>
<td>0.64 0.64 0.64</td>
<td>0.64 0.64 0.64</td>
</tr>
<tr>
<td>4</td>
<td>0.55 0.55 0.55</td>
<td>0.45 0.45 0.45</td>
<td>0.45 0.45 0.45</td>
<td>0.45 0.45 0.45</td>
</tr>
<tr>
<td>6</td>
<td>0.36 0.36 0.36</td>
<td>0.36 0.36 0.36</td>
<td>0.36 0.36 0.36</td>
<td>0.36 0.36 0.36</td>
</tr>
<tr>
<td>8</td>
<td>0.36 0.27 0.27</td>
<td>0.27 0.27 0.27</td>
<td>0.36 0.27 0.27</td>
<td>0.36 0.27 0.27</td>
</tr>
<tr>
<td>10</td>
<td>0.45 0.36 0.27</td>
<td>0.36 0.36 0.27</td>
<td>0.45 0.27 0.27</td>
<td>0.45 0.27 0.27</td>
</tr>
<tr>
<td>12</td>
<td>0.55 0.45 0.36</td>
<td>0.45 0.45 0.36</td>
<td>0.55 0.36 0.36</td>
<td>0.55 0.36 0.36</td>
</tr>
</tbody>
</table>

In Table 2, the results are given for a fault after 5 global multigrid iterations. All global recovery strategies can reduce the delay compared to a do-nothing strategy and can even achieve $\kappa = 0$. On a midsize cluster, Scenario III represents a faulty-healthy ratio of a failure of a single processor, and Scenario I of a complete compute node. Relevant cases for petascale systems will be considered in the next section. We observe a clear advantage in using a global instead of an LR strategy. In the case of the LR strategy, most of the machine stays idle for $n_I$ local multigrid cycles. Only in the case of the two global recovery strategies, DD and DN, do we fully exploit the computing power of the total system. In both global strategies, $n_I$ plays a crucial role in the overall performance. If $n_I$ is too large, we are oversolving the subdomain problems, and if $n_I$ is too small, then the approximation on the faulty subdomain is too inaccurate, and the global performance suffers from the pollution of the local defect into the global subdomain.

To study the proper selection of $n_I$ and $\eta_s$ depending on $k_F$, we consider in Table 3 the situation of a fault after 11 multigrid cycles. In comparison to Table 2, the cycle advantage value $\kappa$ is more sensitive to the choice of $n_I$, in particular, for a small $\eta_s$. This results from the fact that now the local subproblem solver on the faulty domain must counterbalance the smaller residual on the healthy domain. The later the fault occurs, the more powerful the recovery strategy has to be. In particular, without a significantly increased computing power per degree of freedom in the faulty domain compared to the healthy domain, we cannot fully compensate for the fault.

A good choice is to have $n_F$ roughly in the range $[k_F - 2, k_F + n_I]$. This rule of thumb is based on the observation that we need roughly $k_F - 2$ iterations to compensate for the fault in the LR, and on the assumption that our global accuracy does not yet suffer from the reduced communication. For $k_F = 11$ and $\eta_s = 5$, the best choice is $n_F = 10$, i.e., $n_I = 2$ (see Table 3), and for $k_F = 5$ and $\eta_s = 5$, we find that $n_F = 5$, i.e., $n_I = 1$, gives the best results (see Table 2). Moreover, $n_I$ has to be selected with care. First, $\kappa = 0$ can be obtained only for $k_F + n_I \leq k_{free}$. Second, $n_I$ has to be large enough such that on $\Omega_F$ we can compensate for the fault by $n_F = n_I \eta_s$ recovery steps, and third, it has to be small enough such that the decoupling does not deteriorate the global convergence rate. If $n_I$ is too large, then
too many steps without information exchange at the interface between the healthy and faulty subdomains are carried out, and the multigrid method acts as direct solver on the subdomains with inexact boundary data. The DN recovery is more robust than the DD recovery with respect to the choice of \( n_I \), and is the only strategy that achieves in all considered situations a cycle advantage of \( \kappa = 0 \). If the above rule of thumb would require a value larger than 3 or a maximum of 4 for \( n_I \), then one should increase the superman power to obtain better results.

**Remark 2.** Algorithms 2 and 3 presented here are not restricted to successive V-cycle applications, but the ideas can be also realized in a full multigrid (FMG) scheme and preconditioned Krylov solver. A combination of our techniques with such a method can be realized, e.g., following [2].

6. Parallel recovery for single and multiple faults. Now, we investigate both global recovery strategies in a parallel setting on a state-of-the-art petascale system. We study the influence of the algorithmic parameters such as \( \eta_h \) and \( n_I \) but also the influence of the stopping criterion, the fault entries, and fault sizes. Our test system is JUQUEEN,1 an IBM BlueGene/Q system with an overall peak performance of 5.9 petaflop/s, listed in position 13 of the TOP5002 list (47th edition, June 2016). Each of the 28,072 nodes is equipped with 16 cores, and each core can execute up to four hardware threads to help hide latencies. The HHG software is compiled by the IBM XL C/C++ compiler V12.1 using flags `-O3 -qstrict -qarch=qp -qtune=qp` and is linked to MPICH2 version 1.5.

In our experiments, we use up to a quarter of JUQUEEN and execute two of the four hardware threads on a core [37]. We assign one MPI process to each hardware thread and consider, for the moment, failures of MPI processes. In subsection 6.3 node failures also will be presented. Technically, we realize the superman strategy by a logical splitting of the faulty tetrahedron(s) and by employing additional computing power of the size of the faulty processing unit, i.e., MPI processes or complete compute nodes, to perform the recovery. The base meshes for our weak scaling results consist of \( 6 \times (2^l + 1)^3 \) coarse mesh tetrahedra, \( l \in \{1, 2, 3, 4, 5\} \). After refinement to full resolution, each coarse mesh tetrahedron has \( \approx 2.8 \cdot 10^6 \) grid points and is assigned to one hardware thread. In our largest simulations, we use 215,622 threads for \( 6.0 \cdot 10^{11} \) degrees of freedom. The ratio between the faulty and healthy domains decreases from 0.6\% to 4.7\% \( 10^{-4} \)\% in the single-fault case.

In order to accommodate the fact that in the faulty domain \( \Omega_F \), the time for executing a multigrid cycle is possibly shorter compared to the time in the global \( \Omega_I \), we do not fix the number of iterations in the faulty domain a priori to \( n_F = n_I \eta_h \) as in section 5. Instead, we select \( n_F = n_I \eta_h + \Delta n_F \) dynamically such that \( \Delta n_F \) is as large as possible and the run-time max(\( t_I, t_F \)) during the recovery is roughly equal to \( t_I \). We note that for a nonoptimal coarse grid solver, such as the one we use, we expect that \( \Delta n_F \) increases within a weak scaling test series. In the context of section 5, a \( \Delta n_F > 0 \) would correspond to an effective superman that is characterized by an acceleration factor strictly larger than \( \eta_h \). By this modification, the algorithm is more robust with respect to the choice of the coarse grid solver and the available strong scaling headroom. Using a PCG method as coarse grid solver, the number of coarse grid iterations per new level is asymptotically doubled. This occurs for the largest computations in 320 coarse grid PCG iterations in each multigrid cycle. Technically,
the asynchronous execution is realized by the healthy domain sending a signal to the faulty domain just before the last postsmoothing step. On receipt of the signal, the faulty domain then terminates its subtasks and returns to the next synchronization point. Synchronization of the interface values takes place after each iteration in the healthy domain for the DN recovery, and after \( n_I \) iterations in the healthy domain for the DD recovery; cf. Algorithms 3 and 2.

6.1. Single-fault scenarios. Here, we consider a single fault of an MPI process and two different cases for the stopping criterion. In Table 4, we present run-time results for a level-independent stopping criterion for the relative residual \( 10^{-14} \). Although our coarse grid solver is asymptotically nonoptimal, the overall solver is quite fast. For all system sizes, a constant number of 15 multigrid cycles is required. The time-to-solution ranges from 34.17s to 51.07s, yielding a parallel efficiency larger than 65% within a scaling of the problem size by more than three orders of magnitude. The first column displays the total number of unknowns. We specify the extra run-time for a do-nothing execution and illustrate the influence of \( n_I \in \{1, 2, 3, 4\} \) and \( \eta_s \in \{1, 2, 4, 8\} \) on the extra run-time for the two global recovery strategies. In the case of the do-nothing execution, we also specify the number of required multigrid cycles. In the weak scaling, the additional number of multigrid cycles reduces

<table>
<thead>
<tr>
<th>DoF</th>
<th>No rec.</th>
<th>DD strategy ( n_I = 1 )</th>
<th>DN strategy ( n_I = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( \eta_s = 1 )</td>
<td>( \eta_s = 1 )</td>
</tr>
<tr>
<td>( 4.5 \cdot 10^8 )</td>
<td>13.67 (21)</td>
<td>11.46 9.13 2.28 0.00</td>
<td>11.47 9.14 2.29 0.01</td>
</tr>
<tr>
<td>( 2.1 \cdot 10^9 )</td>
<td>11.69 (20)</td>
<td>11.63 7.04 2.42 0.11</td>
<td>11.66 7.05 2.44 0.14</td>
</tr>
<tr>
<td>( 1.2 \cdot 10^{10} )</td>
<td>12.49 (20)</td>
<td>9.90 7.43 2.46 -0.02</td>
<td>9.94 7.47 2.52 0.02</td>
</tr>
<tr>
<td>( 8.2 \cdot 10^{10} )</td>
<td>11.16 (19)</td>
<td>11.07 8.41 0.06 0.02</td>
<td>11.10 8.39 0.11 0.13</td>
</tr>
<tr>
<td>( 6.0 \cdot 10^{11} )</td>
<td>13.59 (19)</td>
<td>6.78 3.48 0.18 0.05</td>
<td>6.82 3.56 0.23 0.09</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DoF</th>
<th>No rec.</th>
<th>DD strategy ( n_I = 2 )</th>
<th>DN strategy ( n_I = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( \eta_s = 1 )</td>
<td>( \eta_s = 1 )</td>
</tr>
<tr>
<td>( 4.5 \cdot 10^8 )</td>
<td>13.67 (21)</td>
<td>11.45 4.56 0.00 0.01</td>
<td>11.47 4.57 0.01 0.03</td>
</tr>
<tr>
<td>( 2.1 \cdot 10^9 )</td>
<td>11.69 (20)</td>
<td>9.31 4.70 0.09 0.10</td>
<td>9.33 4.72 0.10 0.15</td>
</tr>
<tr>
<td>( 1.2 \cdot 10^{10} )</td>
<td>12.49 (20)</td>
<td>9.88 2.46 -0.02 -0.01</td>
<td>9.96 2.51 0.05 0.04</td>
</tr>
<tr>
<td>( 8.2 \cdot 10^{10} )</td>
<td>11.16 (19)</td>
<td>8.30 0.08 0.06 0.02</td>
<td>11.11 2.87 0.13 0.15</td>
</tr>
<tr>
<td>( 6.0 \cdot 10^{11} )</td>
<td>13.59 (19)</td>
<td>6.80 0.09 0.21 0.08</td>
<td>3.49 0.19 0.32 0.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DoF</th>
<th>No rec.</th>
<th>DD strategy ( n_I = 3 )</th>
<th>DN strategy ( n_I = 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( \eta_s = 1 )</td>
<td>( \eta_s = 1 )</td>
</tr>
<tr>
<td>( 4.5 \cdot 10^8 )</td>
<td>13.67 (21)</td>
<td>9.14 -0.01 0.00 0.00</td>
<td>11.47 2.30 0.02 0.04</td>
</tr>
<tr>
<td>( 2.1 \cdot 10^9 )</td>
<td>11.69 (20)</td>
<td>9.31 0.04 0.08 0.11</td>
<td>9.35 2.41 0.11 0.14</td>
</tr>
<tr>
<td>( 1.2 \cdot 10^{10} )</td>
<td>12.49 (20)</td>
<td>7.42 -0.01 -0.02 0.00</td>
<td>9.96 2.54 0.06 0.06</td>
</tr>
<tr>
<td>( 8.2 \cdot 10^{10} )</td>
<td>11.16 (19)</td>
<td>5.54 0.08 0.07 0.04</td>
<td>8.36 0.11 0.15 0.17</td>
</tr>
<tr>
<td>( 6.0 \cdot 10^{11} )</td>
<td>13.59 (19)</td>
<td>3.47 0.13 0.19 0.13</td>
<td>0.13 0.24 0.29 0.26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DoF</th>
<th>No rec.</th>
<th>DD strategy ( n_I = 4 )</th>
<th>DN strategy ( n_I = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( \eta_s = 1 )</td>
<td>( \eta_s = 1 )</td>
</tr>
<tr>
<td>( 4.5 \cdot 10^8 )</td>
<td>13.67 (21)</td>
<td>6.86 2.27 2.28 2.28</td>
<td>9.18 2.30 2.32 2.33</td>
</tr>
<tr>
<td>( 2.1 \cdot 10^9 )</td>
<td>11.69 (20)</td>
<td>6.98 2.37 2.41 2.44</td>
<td>9.37 0.09 0.11 0.15</td>
</tr>
<tr>
<td>( 1.2 \cdot 10^{10} )</td>
<td>12.49 (20)</td>
<td>7.41 -0.01 0.00 0.00</td>
<td>9.98 0.08 0.07 0.07</td>
</tr>
<tr>
<td>( 8.2 \cdot 10^{10} )</td>
<td>11.16 (19)</td>
<td>5.54 0.09 0.09 0.05</td>
<td>8.37 0.13 0.17 0.19</td>
</tr>
<tr>
<td>( 6.0 \cdot 10^{11} )</td>
<td>13.59 (19)</td>
<td>0.10 0.15 0.19 0.15</td>
<td>0.19 0.32 0.39 0.28</td>
</tr>
</tbody>
</table>
from 6 to 4, and the increase in the time-to-solution decreases from 40.04\% to 26.6\%.

For example, for the largest problem with \( 6 \cdot 10^{11} \) unknowns, the time-to-solution increases by 13.59s from 51.07s to 65.06s when no recovery strategy is applied. This effect results from the fact that in the largest experiment only less than 5 \cdot 10^{-4}\% of unknowns are affected by the fault. For the smallest execution, this percentage is 1000 times larger. Thus, the local accuracy in the faulty domain contributes much less to the global one on finer scales.

When increasing \( \eta_s \) (considering one row of DD or DN strategy in Table 4), the time-to-solution can be improved or stays almost constant. In the weak scaling (considering one column of a strategy in Table 4), the recovery strategy profits from the smaller ratio of faulty/healthy domain. In most cases, no extra multigrid cycles compared to the fault-free case are required, and only a small overhead occurs due to the additional synchronization cost, the reinitializing of the faulty domain and the modification of the right-hand side at the interface in the case of the DN strategy. Because the DN strategy needs more synchronization points compared to the DD strategy, it has a slightly larger overhead (< 0.40s versus < 0.22s).

Obviously, the fastest superman with \( \eta_s = 8 \) yields the most robust choice with respect to the parameter \( n_I \) and the selected strategy. But also for \( \eta_s = 4 \), we find in most cases the same qualitative results. The DD strategy is almost as robust as the DN setting with respect to \( n_I \). A recoupling after \( n_I = 4 \) cycles is for the smaller system sizes already too late and leads to a slowdown. In contrast to the previous section, the DD strategy can now profit from the fewer synchronization points compared to the DN approach.

In the weak scaling \( \Delta n_F \) increases for \( \eta_s = 8 \) and \( n_I = 2 \) from 0 to 9 in the DD case, and for the DN strategy, from 0 to 6. This results in an effective superman factor of 12.5 and 11, respectively, instead of \( \eta_s = 8 \). This increase is mainly due to the use of a nonoptimal coarse grid solver. For scalable coarse grid solvers, the increase will be less. However, we note that \( \eta_s = 4 \) results in an effective superman of less than 8. Thus, even a perfectly scalable coarse grid solver does not require a superman factor of more than 8.

For the DN strategy, the choice \( (n_I, \eta_s) = (2, 4) \) or \( (3, 4) \) is quite good, while for the DD strategy one would prefer \( (n_I, \eta_s) = (3, 2) \). Note that as expected, a larger \( \eta_s \) does not deteriorate the time-to-solution.

Next, we study a setup in which a level-dependent stopping criterion is used. In Table 5, we choose a stopping criterion of \( 10^{-8} \) for the smallest test case and reduce it by a factor of 4 in each refinement of the weak scaling. The failure is injected after 4 iterations, and we apply the DN strategy. In the first column, the error given in a properly scaled Euclidean norm between finite element approximation and nodal interpolation is displayed. The required V-cycle number to satisfy the stopping criterion grows from 7 to 10 for a fault-free execution, and for a do-nothing execution from 10 to 11 in the weak scaling. Again, the recovery strategy can achieve the fault-free time-to-solution. The introduced overhead is comparable to that of Table 4. Note that our recovery strategies can also be applied with minor modifications to FMG cycles, which are known for their optimal complexity.

6.2. Multiple-fault scenario. Although the probability distribution for faults is of high relevance in practice (see [30]), it does not significantly influence our recovery strategy. All strategies can be easily extended to a failure of more MPI processes. The performance of the recovery stays robust with respect to the location and entry time.
Table 5
Additional time spans (in seconds) of the DN recovery strategies during weak scaling with a fault after $k_F = 4$ with stopping criterion adjusted to the discretization error.

<table>
<thead>
<tr>
<th>Error</th>
<th>Rate</th>
<th>No fault</th>
<th>No rec.</th>
<th>DN strategy $\eta_s = 2$</th>
<th>DN strategy $\eta_s = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$n_I = 1$</td>
<td>$n_I = 2$</td>
</tr>
<tr>
<td>1.557 · 10^{-6}</td>
<td>$-$</td>
<td>15.73 (7)</td>
<td>6.84 (10)</td>
<td>2.31</td>
<td>0.04</td>
</tr>
<tr>
<td>5.619 · 10^{-7}</td>
<td>1.99</td>
<td>18.62 (8)</td>
<td>4.66 (10)</td>
<td>2.36</td>
<td>0.04</td>
</tr>
<tr>
<td>1.732 · 10^{-7}</td>
<td>2.00</td>
<td>22.30 (9)</td>
<td>2.49 (10)</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>4.846 · 10^{-8}</td>
<td>2.00</td>
<td>24.97 (9)</td>
<td>5.56 (11)</td>
<td>0.06</td>
<td>0.09</td>
</tr>
<tr>
<td>1.291 · 10^{-8}</td>
<td>2.00</td>
<td>34.16 (10)</td>
<td>3.40 (11)</td>
<td>0.11</td>
<td>0.13</td>
</tr>
</tbody>
</table>

In Tables 6 and 7, we test scenarios where two faults occur. From the previous sections, we know that $\eta_s n_I$ has to be selected properly depending on $k_F$. Thus, we do not necessarily use the same $n_I$ for the first and the second fault. This more flexible choice is more important for faults that are strictly separated. Having multiple faults at different locations at the same time does not impose any challenge for the recovery, and thus, we do not report on any results.

In Table 6, we assume a relatively high mean time between faults (MTBF) such that the two recovery steps do not overlap. The first fault occurs after 5 multigrid iterations and the second one after 9 at different locations of the computational domain. For the recovery, we select the DN strategy with $\eta_s = 4$, and test different $(n_I^1, n_I^9)$, where $n_I^5$ and $n_I^9$ denote the number of recovery iterations in the healthy domain for the fault after 5 and 9 iterations, respectively.

Table 6
Additional time spans (in seconds) of the DN recovery strategies during weak scaling with a fault after $k_F = 5$ and $k_F = 9$ iterations.

<table>
<thead>
<tr>
<th>DoF</th>
<th>No rec.</th>
<th>DN strategy $\eta_s = 4$</th>
<th>DN strategy $\eta_s = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(1, 2) (1, 3) (2, 2) (2, 3)</td>
<td>(1, 1) (1, 2) (2, 2) (2, 3)</td>
</tr>
<tr>
<td>4.5 · 10^8</td>
<td>18.35 (23)</td>
<td>0.02 0.03 0.03 0.04</td>
<td>0.02 0.02 0.03 0.02</td>
</tr>
<tr>
<td>2.1 · 10^9</td>
<td>16.33 (22)</td>
<td>0.05 0.06 0.06 0.06</td>
<td>0.06 0.06 0.06 0.08</td>
</tr>
<tr>
<td>1.2 · 10^{10}</td>
<td>17.43 (22)</td>
<td>0.07 0.08 0.09 0.08</td>
<td>0.08 0.09 0.10 0.08</td>
</tr>
<tr>
<td>8.2 · 10^{10}</td>
<td>16.69 (21)</td>
<td>0.16 0.17 0.16 0.17</td>
<td>0.13 0.15 0.18 0.19</td>
</tr>
<tr>
<td>6.0 · 10^{11}</td>
<td>20.64 (21)</td>
<td>0.30 0.33 0.36 0.36</td>
<td>0.35 0.40 0.45 0.47</td>
</tr>
</tbody>
</table>

For $6.0 · 10^{11}$ unknowns, the time-to-solution increases due to the second fault from $51.07 s$ for the fault-free run to $71.71 s$ in case of two faults, and in total, we need 21 multigrid cycles to satisfy the stopping criterion. Further, the additional consumed computation time increases in comparison to the single-fault scenario and now ranges from 53.7% to 40.4% of the fault-free time. We reach the times of the fault-free case with a small overhead (double that of the single fault). Again, $\eta_s = 4$ is sufficient to obtain a perfect recovery for all selected $(n_I^1, n_I^9)$ pairs. In all cases, the recovery is robust with respect to the decoupled solves.

In contrast to the previous setup, we consider in Table 7 a lower MTBF such that the two recovery steps overlap. We inject a fault after 6 and 7 multigrid iterations at MPI processes containing neighboring elements and test the DN case for $\eta_s = 4, 8$ and $n_I^6 = n_I = n_I^7$. If $n_I = 1$, the recovery does not overlap, and recovery can be done as in the setup of Table 6. For $n_I > 1$, the recovery overlaps, and different scenarios possibly occur: The first faulty MPI process is in recovery mode, the first and second faulty processes are in recovery mode, or only the second one is in recovery mode.
Table 7
Additional time spans (in seconds) of the DN recovery strategies during weak scaling with a fault after \( k_F = 6 \) and \( k_F = 7 \) iterations.

<table>
<thead>
<tr>
<th>DoF</th>
<th>No rec.</th>
<th>DN strategy ( \eta_s = 4 )</th>
<th></th>
<th>DN strategy ( \eta_s = 8 )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( n_I = 1 )</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>( 4.5 \cdot 10^8 )</td>
<td>13.71 (21)</td>
<td>4.67</td>
<td>0.13</td>
<td>0.15</td>
<td>2.45</td>
</tr>
<tr>
<td>( 2.1 \cdot 10^9 )</td>
<td>11.65 (20)</td>
<td>2.50</td>
<td>0.18</td>
<td>0.19</td>
<td>2.53</td>
</tr>
<tr>
<td>( 1.2 \cdot 10^{10} )</td>
<td>12.43 (20)</td>
<td>2.64</td>
<td>0.21</td>
<td>0.23</td>
<td>0.24</td>
</tr>
<tr>
<td>( 8.2 \cdot 10^{10} )</td>
<td>11.31 (19)</td>
<td>0.14</td>
<td>0.22</td>
<td>0.24</td>
<td>0.25</td>
</tr>
<tr>
<td>( 6.0 \cdot 10^{11} )</td>
<td>13.73 (19)</td>
<td>0.18</td>
<td>0.28</td>
<td>0.33</td>
<td>0.37</td>
</tr>
</tbody>
</table>

Each of these different stages can be handled in a natural way by the algorithm. As before, the recovery strategy can reduce the time-to-solution to the fault-free run-time plus a small overhead if \( n_I \) and \( \eta_s \) are selected properly. The choice \( n_I = 1 \) requires a superman factor of 8 to fully compensate for the fault.

All our recovery strategies are very robust with respect to the location of the fault and can naturally benefit from a small volume ratio between \( \Omega_F \) and \( \Omega_I \).

6.3. Compute node failure scenario. Finally, we investigate the resilience of the next larger entity, i.e., a compute node, of the processing hierarchy of JUQUEEN. Thus, the loss due to a failure is a factor of 32 larger than in the MPI process failure scenarios of sections 6.1 and 6.2. The faulty domain now consists of 32 tetrahedrons that are connected. The faulty processing unit determines the superman such that the replacements are now complete compute nodes. The results in Table 8 show that the recovery strategies are also robust with respect to node failure.

Table 8
Additional time spans (in seconds) of the DN recovery strategies for a node failure.

<table>
<thead>
<tr>
<th>DoF</th>
<th>Loss</th>
<th>No rec.</th>
<th>DN strategy ( \eta_s = 4 )</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n_I = 2 )</td>
<td>3</td>
</tr>
<tr>
<td>( 1.2 \cdot 10^{10} )</td>
<td>0.7%</td>
<td>15.23 (21)</td>
<td>0.04</td>
<td>0.06</td>
</tr>
<tr>
<td>( 8.2 \cdot 10^{10} )</td>
<td>0.1%</td>
<td>14.18 (20)</td>
<td>0.06</td>
<td>0.10</td>
</tr>
</tbody>
</table>

7. Conclusion and outlook. This paper presents the first insight into constructing a fault tolerant parallel multigrid solver. Hard faults result in a loss of dynamical data in a subdomain. Geometric multigrid solvers are inherently well suited to compensate for such a loss of process state and to reconstruct the data based on a redundant storage scheme for numerical values only along the subdomain interfaces and on using efficient recovery algorithms for the bulk of the data. To recover lost numerical values, local reconstruction subproblems with Dirichlet boundary conditions are solved approximately.

In contrast to simple relaxation schemes or standard Krylov subspace methods, local multigrid cycles can be used to recompute the lost data accurately enough by a small number of local solves. This strategy becomes efficient in terms of both cost and time-to-solution when the local multigrid recovery cycles are accelerated by a superman strategy, as it can be realized by an excess parallelization.

Further, we investigate methods that combine local and global processing in an asynchronous way. A Dirichlet–Dirichlet recovery or Dirichlet–Neumann recovery strategy can be used, both of which iterate in the faulty subdomain and the healthy
subdomain independently, until the local solution has been recovered sufficiently well. Only then are the subdomains reconnected to continue the regular multigrid solution process. Both strategies can further reduce the delay in the case of a fault. For different fault scenarios, we observe that the DN strategy is more robust with respect to the number of global solves before interconnecting. Combined with the superman computing power on the small faulty subdomain, the global recovery techniques can result in a full numerical compensation of the fault while costing no additional computing time. The DD strategy can benefit on fine scales more easily from an increased effective superman factor due to the smaller number of synchronization points compared to the DN strategy. The effective superman factor depends on the choice of the coarse grid solver but can be easily counterbalanced by a proper setting of \( \eta_s \).

The robustness and flexibility of the designed algorithms are tested for single- and multiple-fault scenarios on a state-of-the-art petascale system including large-scale computations with up to \( 6 \times 10^{11} \) unknowns.

Acknowledgment. The authors acknowledge the Institute of Mathematical Sciences of the National University of Singapore, where part of this work was performed.

REFERENCES


RESILIENCE FOR MULTIGRID SOLVERS


[32] G. E. Fagg and J. J. Dongarra, FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world, in Recent Advances in Parallel Virtual Machine and Mes-


