SAND2016-4164C

ULFM-MPI Implementation of a Resilient Task-Based Partial Differential Equations Preconditioner

Francesco Rizzi[*]
Sandia National Laboratories
Livermore, CA, USA
fnrizzi@sandia.gov

Karla Morris
Sandia National Laboratories
Livermore, CA, USA
knmorri@sandia.gov

Khachik Sargsyan
Sandia National Laboratories
Livermore, CA, USA
ksargsy@sandia.gov

Paul Mycek
Duke University
Durham, NC, USA
paul.mycek@duke.edu

Cosmin Safta
Sandia National Laboratories
Livermore, CA, USA
csafta@sandia.gov

Bert Debusschere
Sandia National Laboratories
Livermore, CA, USA
bjdebus@sandia.gov

ABSTRACT

We present a task-based domain-decomposition preconditioner for partial differential equations (PDEs) resilient to silent data corruption (SDC) and hard faults.

The algorithm exploits a reformulation of the PDE as a sampling problem, followed by a regression-based solution update that is resilient to SDC. We adopt a server-client model implemented using the *User Level Fault Mitigation* MPI (MPI-ULFM). All state information is held by the servers, while clients only serve as computational units. The task-based nature of the algorithm and the capabilities of ULFM are complemented at the algorithm level to support missing tasks, making the application resilient to hard faults affecting the clients.

Weak and strong scaling tests up to $\sim 115k$ cores show an excellent performance of the application with efficiencies above 90%, demonstrating the suitability to run at large scale. We demonstrate the resilience of the application for a 2D elliptic PDE by injecting SDC using a random single bit-flip model, and hard faults in the form of clients crashing. We show that in all cases, the application converges to the right solution. We analyze the overhead caused by the faults, and show that, for the test problem considered, the overhead incurred due to SDC is minimal compared to that from the hard faults.

1. INTRODUCTION

The reliability of extreme-scale computing machines is expected to decrease as these systems become increasingly more complex [1, 6, 7]. This is due to a combination of factors including, e.g., the presence of many components characterizing these systems, the variable operational modes (e.g. lower voltage to address energy requirements) and the increasing complexity (e.g. more, smaller transistors). Since failure rates for these post-petascale systems are projected to be in the order of tens of minutes [20], fault tolerance and resilience will become a necessity. Programming models and applications will require a resilient infrastructure to be suitable for fault-free simulations over a large number of nodes for reasonable amounts of time. Assuming complete reliability will be highly unrealistic. This sets the need to develop resilience-aware applications for exascale.

Two main categories of system faults can be identified, namely hard and soft faults [6, 14]. Hard faults can cause partial or full computing nodes to fail, or the network to crash. According to [18], the majority of failures affecting application users involves single node failures in large PC clusters. Currently, application checkpoint-restart (C/R) is the most commonly used tool for dealing with hard faults [6]: it involves killing all the remaining processes of a program execution, and restarting the program from the most recent global snapshot of the application. In future extreme scale systems, it might not work as well because the time for C/R will exceed the mean time to failure [6, 16, 7], and it can lead to substantial overhead depending on the simulation size [3]. An interesting work trying to address these scaling issues has been developed in [21]. The authors explore an emerging resilient programming model called Local Failure Local Recovery (LFLR) providing application developers the ability to recover locally and continue application execution when a process is lost. This study is an example fitting the general framework mentioned above, i.e. designing resilience-aware applications.

Silent errors are more subtle than hard faults because they can go undetected and do not kill the running application, e.g. in the case of silent data corruption (SDC). Soft faults cause incorrect arithmetic or storage. The key feature of silent errors is that, being undetected, there is no opportunity for an application to directly recover from the fault when it occurs. Sample studies of the effects of SDC on iterative solvers can be found in [4, 10, 12, 11].

This work presents a task-based domain-decomposition preconditioner for partial differential equations (PDEs) that is resilient to SDC and hard faults. The problem is reformulated such that the PDE solver is reduced to a number of independent tasks to favor concurrency and parallelism.

[*]Corresponding Author

FTXS'16, May 31, 2016, Kyoto, Japan.
© 2016 ACM. ISBN 978-1-4503-4349-7/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2909428.2909429

Resilience to SDC is achieved at the algorithm level by exploiting a reformulation of the PDE as a sampling problem, followed by a solution update through data manipulation based on robust regression yielding resilience to SDC. This is complemented by supporting resilience to hard faults, using a server-client model (SCM) implemented using the *User Level Fault Mitigation* MPI (MPI-ULFM), a fault tolerance capability proposed for the MPI standard that enables a fault-tolerant MPI framework. All state information is held by the servers, while clients only serve as computational units. The server-client programming model provides a task-based infrastructure that can potentially address some of the concerns related to energy consumption and resiliency at extreme-scale. The task-based nature of the algorithm and the capabilities of ULFM are complemented at the algorithm level to support missing tasks, making the application resilient to hard faults affecting the clients.

ULFM provides three main capabilities in relation to fault tolerance: the ability to continue the execution through process failure, the detection and notification of process failure, and revocation and correction of a broken communicator. Among these, revoking and correcting a broken communicator becomes increasingly more expensive as the size of the target communicator increases. The SCM allows us to avoid using these operations, because if a client crashes, its information and local communicator do not need to be reconstructed, since the server can simply continue with the clients that are alive.

Our application can be seen as a preconditioner that will enable today's solvers to be used effectively on future architectures by operating on subdomain levels. Strong and weak scalability tests run up to $\sim 115K$ cores show a parallel efficiency above 90%. We investigate the effects of SDC in the form of random bit-flips, and hard faults injected as client crashing, and quantify their impact on the overhead of target stages of the application as well as the total overhead of the application.

The paper is organized as follows. In § 2, we describe the mathematical formulation; in § 3, we present the implementation details; in § 4, we discuss the results, focusing on the scalability § 4.1, and resilience § 4.2. Finally, in § 5 we present the conclusions.

2. MATHEMATICAL FORMULATION

This work focuses on 2D elliptic PDEs of the form

$$\mathcal{L}y(\mathbf{x}) = g(\mathbf{x}), \tag{1}$$

where $\mathcal{L}$ is an elliptic differential operator, $g(\mathbf{x})$ is a given source term, and $\mathbf{x} = \{x_1, x_2\} \in \Omega \subset \mathbb{R}^2$, with Ω being the target domain region. We focus on Dirichlet boundary condition $y(\mathbf{x})|_{\mathbf{x}\in\Gamma} = y_\Gamma$ along the boundary Γ of domain Ω. The discussion of 1D elliptic PDEs can be found in [19]. Elliptic equations are chosen because they represent a fundamental problem in physics. Extension to parabolic and hyperbolic problems is in progress.

Figure 1 shows a high-level schematic of the algorithm's work-flow. The first step involves the *discretization* of the computational domain, which is arbitrary, potentially heterogeneous across the domain, e.g. uniform, or non-uniform rectangular grid, or a finite-element triangulation, etc.

The second step is the *partitioning* stage, where the target 2D domain, Ω, is divided into a grid of $n_1 \times n_2$ *overlapping* regions (or subdomains), with n_k being the number of sub-

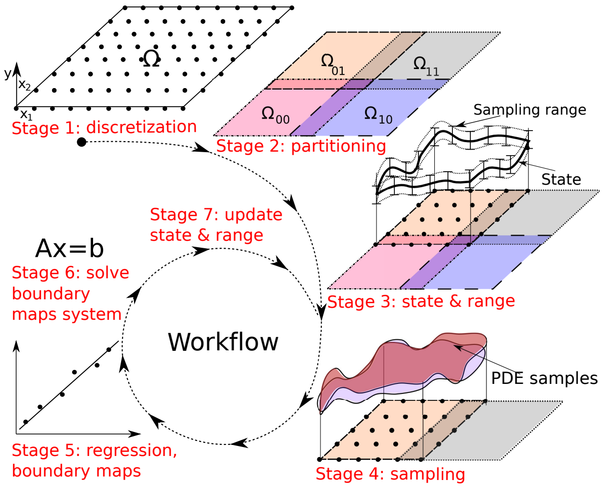

Figure 1: Schematic of the work-flow of the algorithm. For clarity, starting with stage 3 we only show the steps for Ω_{01} but the same operations are applied to all subdomains.

domains along the k-th axis. The size of the overlap does not need to be equal and uniform among all partitions, and can vary across the domain. The partitioning stage yields a set of $n_1 \times n_2$ subdomains Ω_{ij}, and their corresponding boundaries $\Gamma_{s_{ij}}$, for $i = 0, \ldots, n_1 - 1$, and $j = 0, \ldots, n_2 - 1$, where $\Gamma_{s_{ij}}$ represents the boundary set of the ij-th subdomain Ω_{ij}.

We define as our object of interest the set of solution fields along the boundaries, which we denote $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}}$ for $i = 0, \ldots, n_1 - 1$, and $j = 0, \ldots, n_2 - 1$. Due to the overlapping, each subdomain Ω_{ij} includes a set of *inner* boundaries, $\Gamma^{in}_{s_{ij}}$, i.e. the parts of the boundaries belonging to the intersecting (neighboring) subdomains that are contained within Ω_{ij}. The core of the algorithm relies on exploiting within each subdomain Ω_{ij} the *map* relating the solution at the subdomain boundaries, $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}}$, to the solution along the inner boundaries, $y(\mathbf{x})|_{\mathbf{x}\in\Gamma^{in}_{s_{ij}}}$. These maps can be written compactly as

$$y(\mathbf{x})|_{\mathbf{x}\in\Gamma^{in}_{s_{ij}}} = f^{(ij)}\left(y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}}\right), \tag{2}$$

for $i = 0, \ldots, n_1 - 1$, and $j = 0, \ldots, n_2 - 1$. The system of equations assembled from these *boundary-to-boundary maps* collected from all subdomains, combined with the boundary conditions on the full domain $y(\mathbf{x})|_{\mathbf{x}\in\Gamma}$, yields a fixed-point problem of the form

$$\mathbf{y}(\mathbf{x}) = \mathcal{F}\mathbf{y}(\mathbf{x}), \tag{3}$$

where $\mathbf{y}$ represents the vector of the solution values at all subdomains boundaries. This problem is only satisfied by the true solution. We remark that these boundary maps $f^{(ij)}$ relate the y-values, since they are built from the restrictions of the subdomain solutions at the corresponding boundaries.

In this work, rather than solving Eq. (3) directly, we construct *approximations* (or surrogates) of the boundary-to-boundary maps, which we call $\tilde{f}^{(ij)}$. One of the main features of the algorithm is that the construction of these maps can be done for each subdomain *independently* from all the others. This allows us to satisfy data locality, which is one of the main factors contributing to scalability on extreme scale machines. To build these surrogate maps, given a current state of the solution at the subdomains boundaries, we use a sampling strategy that involves solving the target PDE equation locally within each subdomain for sampled values of the boundary conditions on that subdomain, see stage 3 in Figure 1. These samples are used within a regression approach to infer the approximate boundary-to-boundary maps. For non-linear problems the maps are non-linear, while for linear PDEs the boundary maps are linear [19]. Following the construction of the surrogate boundary-to-boundary maps, we can then solve the system of boundary maps, namely the *approximate* version of the fixed point system in Eq. (2), which provides us with the new solution state at all the subdomains boundaries and represents an approximation of the true solution. An important measure of the accuracy of the current solution $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}}$ is the *residual*

$$\mathbf{z} = \mathcal{F}\mathbf{y} - \mathbf{y}, \tag{4}$$

which can be computed by extra subdomain solves using boundary conditions defined by the current solution $\mathbf{y}$, and subtracting the corresponding current solutions $\mathbf{y}$ from the resulting values at all boundaries. Given the fixed-point problem in Eq. (3), the residual (4) vanishes if the current solution $\mathbf{y}$ is the exact solution. In the case of linear PDEs, because the boundary-to-boundary maps are linear, and assuming that all the regressions complete successfully, the algorithm converges in one iteration.

The construction of the boundary-to-boundary maps plays a key role for ensuring resilience against potential silent data corruption (SDC) affecting the PDE samples. As shown in [19], when inferring linear maps, using a ℓ_1-noise model one can seamlessly filter out the effects of few corrupted data. The ℓ_1 noise model yields the solution with as few non-zero residuals as possible. Under the assumption that faults are rare, the inferred maps will fit the non-corrupted data exactly while effectively ignoring the corrupted data. In the present work, we employ an iteratively re-weighted least squares (IRLS) method, which is effectively equivalent to a ℓ_1 minimization [8].

3. IMPLEMENTATION DETAILS

We have developed a parallel, C++ implementation of the algorithm using a server-client model (SCM). This section describes the SCM, its resilience properties, and how we implement each stage of the algorithm to exploit its features.

3.1 Server-Client Model

To support resilience to hard faults, we exploit a server-client programming model, and rely on the *User Level Fault Mitigation* MPI (MPI-ULFM)[1] [2], a fault tolerance capability proposed for the MPI standard enabling a fault-tolerant MPI framework.

The SCM structure involves a set of servers, each connected to the same number of clients for resource balancing. A schematic of the SCM structure is shown in Figure 2. A client is defined as a set of MPI processes, and is designed solely to accept and perform work without any assumption on its reliability. There are three main types of communicators involved. One communicator to include all servers, allowing them to communicate between each other. One

[1]http://fault-tolerance.org/

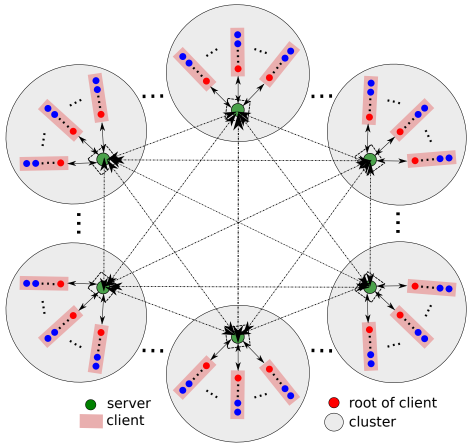

Figure 2: Schematic of the server-client structure.

communicator is defined for each cluster, including a server and the root MPI ranks of all the clients associated with that server. This cluster communicator is used by the servers to interface with the root ranks of the clients. This allows to optimize communication, since the client's root rank receives work from the server, and distributes it among the children ranks within that client. This approach can be more efficient than having the server communicating a task to each child MPI rank in a client. One example is the case where all ranks of a client live in the same node, so that one can exploit in-node parallelism and faster memory access. Lastly, each client owns a communicator used for all the intra-client communication. The clusters and all the clients are independent, i.e. clients are only visible to the server within a given cluster, and at any given time, each client is handling a different work unit. This design choice allows a client to fail without affecting others.

The state information is distributed among the servers, which are assumed to be highly resilient (safe or under a "sandbox" model implementation). The sandbox model assumed for the servers can be supported by software, hardware or a combination of both. Software support can be accomplished via data redundancy and strategic synchronization [17], [5], [13]. Hardware support involves the variable levels of resilience that can be allowed within large computer systems. The servers are responsible for generating work in the form of tasks, dispatching them to their pool of available clients, as well as receiving and processing them.

Separating the state from computation enables resilience to hard faults. Clients crashing do not affect the state information safely owned by the servers, but only translate into missing tasks. Besides, this has to be complemented by designing an application/algorithm that can handle missing data.

3.2 Algorithm Implementation

As described in § 2, the algorithm comprises four main stages: sampling, regression, boundary maps system solve, and updating. Sampling and regression can be performed independently and concurrently across all subdomains. As such, they are implemented in the form of tasks executed by the clients, and are therefore susceptible to the failures occurring on the clients. On the other hand, the fixed-point solve of the boundary-to-boundary maps system and the up-

Table 1: Scalability tests.

	Weak	Strong
Subdomains	18^2, 24^2, 30^2, 42^2, 48^2, 60^2	42^2
Subdomain size	180^2	180^2
Servers	36, 64, 100, 196, 256, 400	49, 98, 196, 441
Clients/server	64	64
Size of client	4 MPI ranks	4 MPI ranks
Total Cores	9252, 16448, 25700, 50372, 65792, 102800	12594, 25186, 50372, 113337

dating of the subdomains are executed by the servers, since they fully own the state information. The system of equations built from the boundary maps is much smaller than the original discretized PDE system over the full domain grid, and so it fits on a small number of servers. Moreover, the servers are assumed to be "sandboxed", allowing us to circumvent any potential data corruption during these two operations. This design choice aligns with the concept of selective reliability [4], where some parts of the algorithm are assumed to be handled in a more reliable manner than others.

4. RESULTS

All the results presented below are based on the following 2D linear elliptic PDE

$$\frac{\partial}{\partial x_1}\left(k(\mathbf{x})\frac{\partial y(\mathbf{x})}{\partial x_1}\right) + \frac{\partial}{\partial x_2}\left(k(\mathbf{x})\frac{\partial y(\mathbf{x})}{\partial x_2}\right) = g(\mathbf{x}), \qquad (5)$$

where $\mathbf{x} = \{x_1, x_2\}$, the field variable is $y(x_1, x_2)$, $k(x_1, x_2)$ is the diffusivity, and $g(x_1, x_2)$ is the source term. This PDE is solved over a unit square $(0, 1)^2$, with homogeneous Dirichlet boundary conditions. The diffusivity and source fields are defined as

$$k(x_1, x_2) = 8.0 * \exp(-d(x_1, x_2)/0.025) + 2.0, \qquad (6)$$
$$g(x_1, x_2) = 2.0 * \exp(-d(x_1, x_2)/0.050) - 1.0, \qquad (7)$$

where $d(x_1, x_2) = (x_1 - 0.35)^2 + (x_2 - 0.35)^2$. To solve the above PDE within each subdomain, we employ a structured grid and second-order finite differences to discretize Eq. (5). The resulting linear system stemming from the finite-difference discretization is solved using AztecOO, which is part of Trilinos [15].

4.1 Nominal Scalability

Weak and strong scaling have been tested using the parameters listed in Table 1. All the runs were completed on Edison (NERSC)[2], a Cray XC30, with Peak performance of 2.57 Petaflops, Cray Aries high-speed interconnect with Dragonfly topology with approximately 8GB/sec MPI bandwidth. The scalability has been performed without any faults, and relying on the native Cray-MPICH. Ongoing work is the scalability on Edison using ULFM-MPI.

Figure 3 shows the weak and strong scaling results. As shown in Table 1, we set up the weak scaling by fixing the number of clients per server and the amount of data owned by each server, while increasing the problem size by adding

[2]http://www.nersc.gov

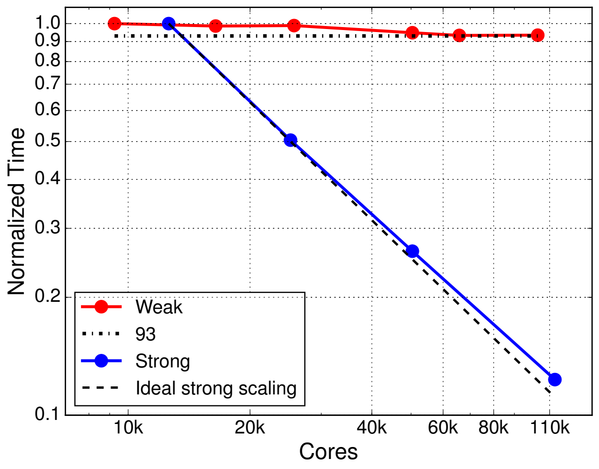

Figure 3: Nominal weak and strong scaling results.

more clusters. This design imposes no constraint on the problem size, since larger problems can be tackled by simply adding more clusters. Figure 3 shows an excellent weak scaling with efficiency settling around 93% up to $\sim 105K$.

Strong scaling is based on fixing the problem size, and testing how the application performs as we increase the computing power. In this work, we increase the computational power by increasing the number of clusters used in the SCM. Figure 3 shows strong scaling results up to $\sim 115k$ cores, which is close to Edison's full capacity ($\sim 133k$). The results are excellent, only slightly deviating from the ideal linear scaling trend superimposed with a dashed line. This result confirms that despite the task-based nature of the SCM, the application is not limited by communication, and is well-suited for running at large scale.

4.2 Resiliency

In this section, we demonstrate the resilience to SDC and hard faults, and discuss the associated overhead. SDC are assumed to affect numerical data used in the algorithm, i.e. we exclude other types of faults e.g., in data structures or control flow [9]. We assume SDC to be caused by bit-flips so the results below are based on a random bit-flip model.

Both SDC and hard faults are only injected in the clients, which is consistent with the SCM described previously, where servers are assumed to be reliable, while no assumption is made on the reliability of the clients. Hard faults are assumed to occur in the form of clients crashing. The advantage of the SCM we are proposing is that when a client crashes, we do not need to call expensive collective procedures of ULFM to rebuild the client and fix communicators, because the server can simply continue the execution using only the clients that are alive. From a practical standpoint, our implementation is based on each server probing the corresponding cluster communicator using `MPI_ANY_SOURCE` to assess whether a new message is arriving from one of the clients. If a client crashes, despite there are other pending messages, the probing loop does not work because ULFM reports the same error continuously (`MPI_ERR_PROC_FAILED`). This problem can be solved by acknowledging the failure using the ULFM function `MPI_COMM_FAILURE_ACK`, and then continue the normal probing loop. This allows us to avoid the need to reconstruct the broken client.

Leveraging a *selective reliability* approach [4, 9, 10, 12, 11], which lets algorithm developers isolate faults to certain parts of the algorithm, SDC are only injected during the sampling stage, while hard faults can hit both sampling and regression.

4.2.1 Fault Injection and Model

To inject faults, we leverage the task-based nature of our algorithm by choosing the number of faults as a percentage of the nominal number of tasks to execute. This is done by randomly selecting the set of task IDs off-line. This eliminates any dependency between faults' occurrence and the execution time, since the latter is machine-dependent.

For hard faults, when a client receives a black listed task ID, the processes owning the MPI ranks within that client are killed. This is possible because of the ULFM-MPI implementation, which allows actual processes to be killed.

For soft faults, if a task is in the black list, we adopt the following procedure: we draw a value, u, from a standard uniform distribution, and if $u \leq 0.5$, the task data is corrupted *before* the execution; if $u > 0.5$, the task data is corrupted *after* its execution. This mimics corruptions occurring when tasks are being transmitted to and from a client, as well as those happening during execution. If a task is corrupted before the execution, this translates into corrupting all boundary conditions pertaining to that sampling task, since the boundary conditions are the only information needed to solve the elliptic PDE. If a task is corrupted after execution, this translates into corrupting all points in the solution, which means that even the inner points of a subdomain can be affected.

4.2.2 Handling Faults

To guarantee the algorithm's resilience to SDC and hard faults affecting the sampling stage, the key condition to be satisfied is that within each subdomain, the number of uncorrupted collected PDE samples has to be greater than the minimum set needed to have a mathematically well-posed regression problem. This can be achieved by generating, within each subdomain, more PDE samples than the minimum set using an oversampling factor, $\rho > 1$, such that the total target number of samples to generate is $N = \rho N^s_{nom}$, where N^s_{nom} is the number of samples needed for the no-fault case. In the present work, the oversampling is chosen as the minimum value to mitigate the effect of the faults. This is possible because we inject a known number of faults. In general, the oversampling can be set based on estimating the reliability of the machine, and inferring the likelihood of faults occurring. During the sampling stage, we apply a filter on the task data returned to the server to check that it is within the interval $(-100, 100)$ before the data is stored within the corresponding subdomain. This interval is arbitrary, but can be estimated by either a domain expert or by known physical bounds on the solution. This is the only active "filter" that is needed by the application. Any other corruption during the sampling does not need to be actively detected, since it is seamlessly filtered out thanks to the mathematical model used in the regression.

As previously mentioned, during the regression only hard faults are injected. Hence, a client crashing yields the loss of regression tasks. To overcome this effect, the server keeps track of the regression tasks that have not come back, and resubmits them until they return.

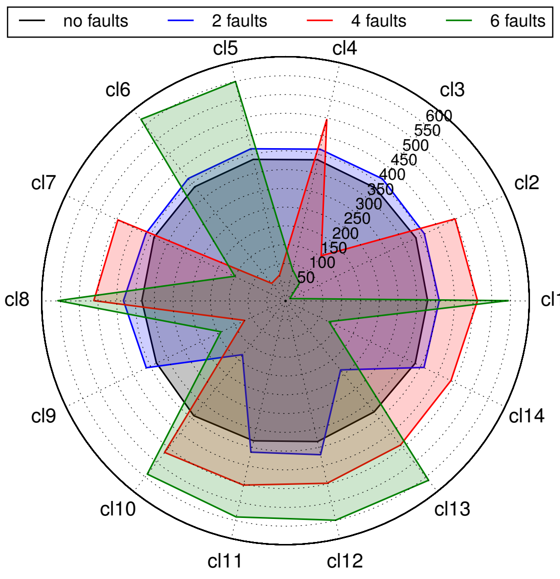

Figure 4: Sample results of the clients' work-load distribution for various hard failures.

4.2.3 Test Problem and Execution

Resilience is tested using the PDE in Eq. (5), solved over a structured uniform grid with 251^2 grid points over the unit square domain. We partition the domain using $n_1 = n_2 = 3$ subdomains, with an overlapping of 10 grid cells between neighboring subdomains. This setting yields a total of 9 subdomains, each with a local grid of $\sim 90^2$ grid points. Nominally, this problem involves $N^s_{nom} = 3249$ sampling tasks, and $N^r_{nom} = 2136$ regression tasks. The SCM structure involves a single server holding the data, i.e. subdomains, and uses 14 clients each made of 2 MPI ranks.

To explore the impact of SDC, we explore three percentages, namely 0.25%, 0.5%, and 1.0%. For hard faults, we consider cases with 2, 4 and 6 clients failing. These cases correspond, respectively, to 14%, 28% and 42% of the computational power. The number of failures chosen is admittedly high, because our goal is to test the application under various extreme scenarios.

The results are presented in two parts: first, we focus our attention on analyzing the effect of hard faults only; second, we discuss the results obtained when both soft and hard faults can occur during the execution. For each scenario, we run an ensemble of 100 runs to have a statistically meaningful result, and extract the main statistics.

4.2.4 Resilience Analysis

Figure 4 shows a comparison of the clients' workload for a sample run with no-faults, and runs with 2, 4 and 6 hard faults randomly happening during the sampling stage. The plot displays along the angular direction the client name, and the data reported indicates the total number of tasks being handled during the simulation. For the no-fault case, the workload is fairly uniform: each client handles nearly the same number of tasks. As the number of faults increases, the asymmetry becomes increasingly more evident. As expected, increasing the number of faults causes the clients that are alive to handle more and more tasks to compensate for those that are dead. Even in the cases with faults, the workload remains fairly uniform among the clients that are alive.

Hard Faults Only. Figure 5 summarizes the main statistics from the ensemble runs performed when only hard faults are injected. Specifically, we show three radar-plots corresponding to hard faults hitting only the sampling stage (a), only the regression (b), and both (c). Each plot displays the average value over the replicas of key quantities in the following counter clock-wise order: the total, regression and sampling overhead with respect to the no-fault case, the percentage of failed clients, and the boolean value identifying convergence. In all cases, all the runs converged to the correct answer, proving that the application is resilient to hard faults. Convergence is verified by checking that the root-mean-square error of the residual in Eq. (4) is lower than a specified threshold.

Figure 5a shows that when hard faults hit the sampling stage, the regression is substantially more affected than the sampling itself. This can be explained as follows: while during the sampling hard faults occur at random and, on average, are distributed across the full stage, the entire regression stage has to be completed with reduced computational power. The total overhead is a weighted average of the overhead from the sampling and the one of the regression. In this case, in order to make the application resilient, we only need to run as many additional samples per subdomain as the number clients that crash. For the case with 2 clients failing, for instance, the overhead incurred in the sampling is only about 5%, and the regression overhead is about 14%. The total overhead thus end up being about 11%, despite the fact that this case corresponds to a 14% loss of the computational power.

Figure 5b shows the results when hard faults occur only during the regression. In this case, there is no need to have any oversampling, yielding no overhead during the sampling stage. All the overhead is within the regression stage. Again, the results show the resilience of the application since it converges in all cases. In this scenario, we highlight the relationship between the percentage of failures and the total overhead of the application: for 14%, 28% and 42% loss of clients, we observe a total overhead of about 8%, 14% and 20%, respectively, which is nearly half in all three case. This is because the sampling stage has no overhead, since all faults occur during the regression.

Figure 5c shows the results obtained when hard faults are injected during both sampling and regression. The number of faults hitting either stage is drawn at random, thus involving scenarios where all faults hit the sampling, or just the regression, or both. The results shown in panel (c) can be seen as a combination of panel (a) and (b). It is interesting to note how losing 14%, 28% and 42% of the clients yields, respectively, a total overhead of 8%, 19% and 30%.

The three plots show, as expected, that the best case scenario is when all faults are limited during the regression stage because full computational power is available for a longer part of the simulation. Overall, an interesting observation is that in all cases, the total overhead is equal or lower than the loss of computational power.

Hard Faults and SDC. Figure 6 summarizes the main results from the ensemble runs performed when both hard faults and SDC are injected. Each plot displays the average

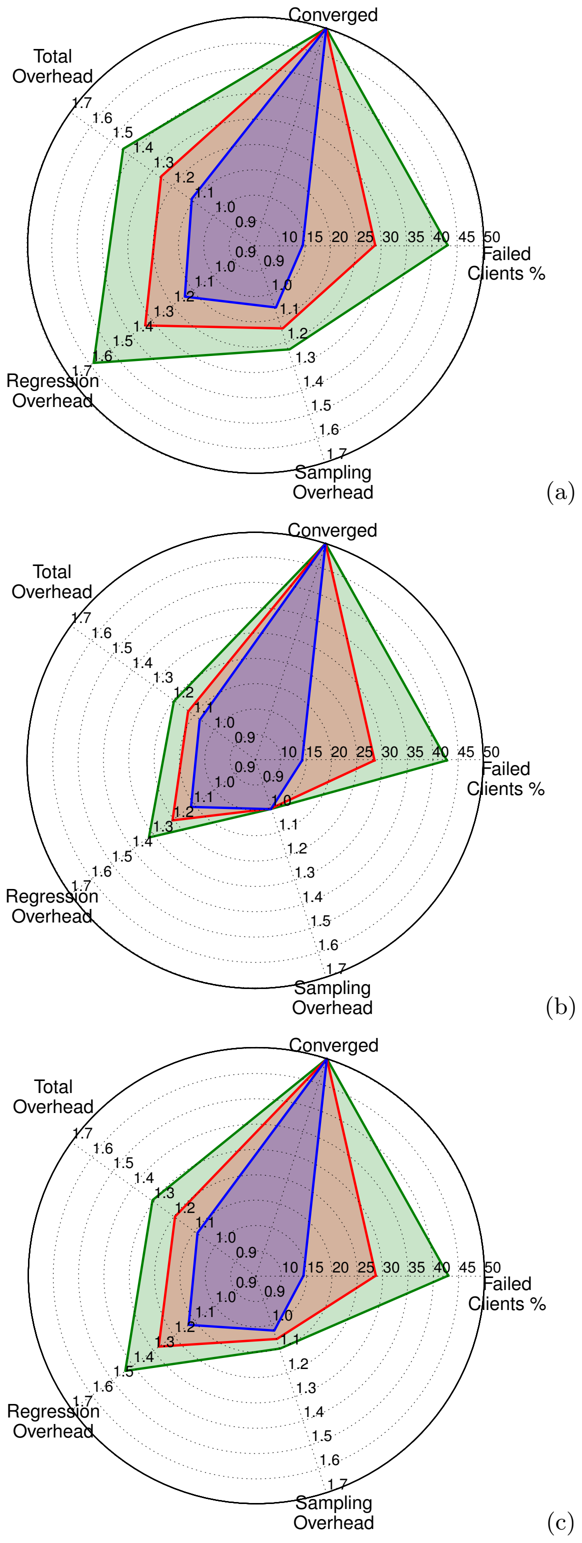

Figure 5: Results obtained from the ensemble runs performed for the case with *hard* faults only. The radar-plots correspond to the following cases: (a) hard faults during sampling; (b) hard faults during regression; (c) hard faults during sampling and regression.

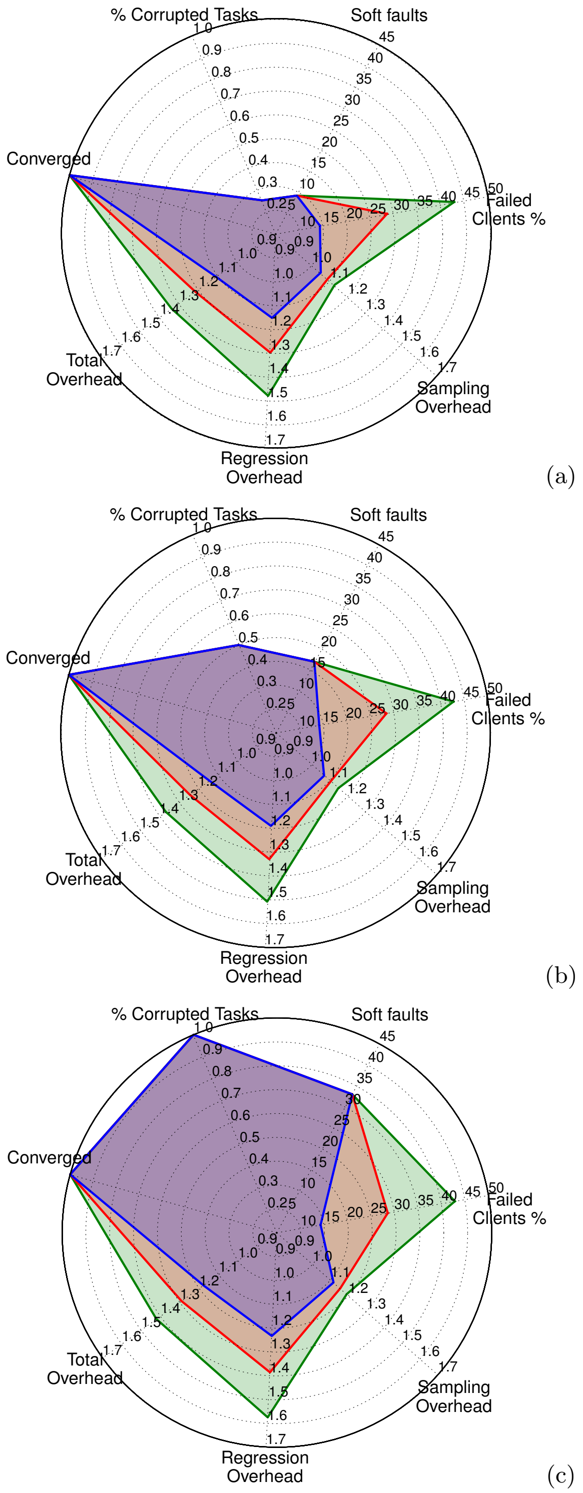

Figure 6: Statistical results obtained from the ensemble runs performed for the case with both *hard* and *soft* faults. The radar-plots correspond to the various levels of SDC injected, namely 0.25 % **(a),** 0.5 % **(b), and** 1.0 % **(c).**

value over the replicas of target quantities in the following counter clock-wise order: the percentage of sampling tasks affected by SDC, the boolean value identifying convergence, the total, regression and sampling overhead with respect to the no-fault case, the percentage of failed clients, and the actual number of tasks affected by SDC. We recall that SDC are injected during the sampling stage, while hard faults are injected during both sampling and regression. In this scenario, the key to make the application resilient is to consider an oversampling accounting for both the present of hard and soft faults. For example, for 0.25% SDC and 4 hard faults, this means generating about 13 more samples per subdomain with respect to the no-fault case.

First, all runs converge, demonstrating the resilience even when both SDC and hard faults occur together. Second, for a given number of hard faults, the results show an interesting trend for the overhead in the sampling and regression stages. For example, when the number of hard faults is 4, a nearly *four-fold* increase in the number of SDC from 9 to 33 causes the sampling overhead to only increase from 9% (Figure 6a) to about 15% (Figure 6c). Similarly, the regression overhead only increases from 30% to about 38%. This yields the total overhead to only increase from 21% to 28%. A similar argument can be made for the other cases. The total overhead of our application is thus only minimally sensitive to SDC affecting the sampling stage. These results demonstrate how well the application can tolerate soft and hard faults.

5. CONCLUSIONS

This work discussed algorithm-based resilience to silent data corruption (SDC) and hard faults in a task-based domain-decomposition preconditioner for PDEs.

The algorithm involves four main steps: first, the domain of the PDE is split into overlapping subdomains; second, the PDE is solved on each subdomain for sampled values of the local current boundary conditions; third, the resulting subdomain solution samples are fed into a regression step to build boundary-to-boundary maps; finally, the intersection of these maps yields the updated state at the subdomain boundaries.

Support to hard faults is enabled by relying on a server-client model implemented using ULFM-MPI where all state information is held by the servers, while clients are designed solely as computational units. The application and parallel implementation was tested for weak and strong scaling up to $\sim 115K$ cores, showing an efficiency greater than 90%.

We used a 2D elliptic PDE, a fault model based on random single bit-flip and target reliability assumption to show that the application is resilient to SDC injected during the sampling stage, and hard faults occurring during sampling and regression. The resilience to SDC is inherent in the algorithm thanks to the robust regression based on the ℓ_1 regression model, allowing to seamlessly filter out the effect of corrupted data. The results are promising. They show that the overhead due to the presence of SDC is minimal compared to the contribution due to hard faults. Moreover, the total overhead observed is never greater than the one estimated by accounting for the loss of computational power.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Numbers 13-

016717. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

7. ADDITIONAL AUTHORS

Additional authors: Olivier LeMaitre (LIMSI, France, email: `olm@limsi.fr`) and Omar Knio (Duke University, Durham, NC, USA, email: `omar.knio@duke.edu`)

8. REFERENCES

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead, 2008.

[2] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *Int. J. High Perform. Comput. Appl.*, 27(3):244–254, Aug. 2013.

[3] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, Apr 2009.

[4] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *ArXiv e-prints*, June 2012.

[5] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *ArXiv e-prints*, June 2012.

[6] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, oct 2009.

[7] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[8] I. Daubechies, R. DeVore, M. Fornasier, and C. S. Güntürk. Iteratively reweighted least squares minimization for sparse recovery. *Communications on Pure and Applied Mathematics*, 63(1):1–38, 2010.

[9] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of SDC on the GMRES iterative solver. *CoRR*, abs/1311.6505, 2013.

[10] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of sdc on the gmres iterative solver. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1193–1202, Washington, DC, USA, 2014. IEEE Computer Society.

[11] J. Elliott, M. Hoemmen, and F. Mueller. A numerical soft fault model for iterative linear solvers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 271–274, New York, NY, USA, 2015. ACM.

[12] J. J. Elliott, M. F. Hoemmen, and F. Mueller. *Tolerating Silent Data Corruption in Opaque Preconditioners.* Apr 2014.

[13] C. Engelmann and T. Naughton. Toward a performance/resilience tool for hardware/software co-design of high-performance computing systems. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 960–969, Oct 2013.

[14] R. Gupta, K. Iskra, K. Yoshii, P. Balaji, and P. Beckman. Introspective fault tolerance for exascale systems. Technical report, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, 2012.

[15] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[16] D. Li, J. S. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 57:1–57:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[17] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. *SIGOPS Oper. Syst. Rev.*, 42(2):265–276, Mar. 2008.

[18] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[19] K. Sargsyan, F. Rizzi, P. Mycek, C. Safta, K. Morris, H. Najm, O. L. Maître, O. Knio, and B. Debusschere. Fault resilient domain decomposition preconditioner for pdes. *SIAM Journal on Scientific Computing*, 37(5):A2317–A2345, 2015.

[20] M. Snir, W. Gropp, , and P. Kogge. Exascale Research: Preparing for the Post-Moore Era. Technical report, Computer Science Whitepapers, 2011.

[21] K. Teranishi and M. A. Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 51:51–51:56, New York, NY, USA, 2014. ACM.