SAND2015-0947C

Partial Differential Equations Solver Resilient to Soft and Hard Faults

F. Rizzi*
Sandia National Laboratories
Livermore, CA, USA
fnrizzi@sandia.gov

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

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

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

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

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

ABSTRACT

In this paper, we present a domain-decomposition preconditioner for the solution of partial differential equations (PDEs) that is resilient to both soft and hard faults. The algorithm involves the following main steps: first, the target domain of the PDE is split into overlapping subdomains; second, the target 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 maps between the subdomains' boundary conditions; finally, the intersection of these maps yields the updated state at the subdomain boundaries. This reformulation allows us to recast the problem as a set of independent tasks. We rely on an asynchronous server-client framework, where one or more servers, which are assumed to be "safe", hold the data, while the clients ask for tasks and execute them. This framework provides resiliency to hard faults such that if a client crashes, it stops asking for work, and the server simply distributes the work among all the other clients that are still alive. Erroneous subdomain solves (e.g. due to soft faults) appear as corrupted data, which is either rejected if that causes a task to fail, or is seamlessly filtered out during the regression stage through a suitable noise model. Three different types of faults are modeled: hard faults modeling nodes (or clients) crashing; soft faults occurring during the communication of the tasks between server and clients; and soft faults occurring during task execution. These faults are modeled using a Poisson process defined by a failure rate extracted from literature. We demonstrate the resiliency of the approach for a 2D elliptic PDE, and explore the effect of the faults at various failure rates.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems—*Client/server, Distributed applications*; G.1.8 [**NUMERICAL ANALYSIS**]: Partial Differential Equations—*Domain decomposition methods, Finite difference methods, Elliptic equations*; G.1.0 [**NUMERICAL ANALYSIS**]: General—*Numerical algorithms, Parallel algorithms*; G.1.3 [**NUMERICAL ANALYSIS**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods)*; D.2.3 [**SOFTWARE ENGINEERING**]: Coding Tools and Techniques—*Object-oriented programming*; B.4.5 [**INPUT/OUTPUT AND DATA COMMUNICATIONS**]: Reliability, Testing, and Fault-Tolerance—*Hardware reliability*; B.8.1 [**PERFORMANCE AND RELIABILITY**]: Reliability, Testing, and Fault-Tolerance; D.4.1 [**OPERATING SYSTEMS**]: Process Management—*Multiprocessing, multiprogramming, multitasking, Scheduling, Synchronization*; C.2.0 [**COMPUTER-COMMUNICATION NETWORKS**]: General—*Data communications*

*Corresponding Author

1. INTRODUCTION

As computing platforms evolve towards exascale, several key challenges are arising related to resiliency, power, memory access, concurrency and heterogeneous hardware, see [8, 1, 9, 4, 5] and references therein. There is no consensus or clear idea yet on what a "typical" exascale architecture might look like [1]. One of the main concerns is understanding how the hardware will affect future computing systems in terms of reliability, as well as communication and computational models, and which ones will emerge to become the main reference for exascale.

Exascale simulations are expected to rely on millions of nodes, and also take advantage of local thousand-fold concurrency through cores or threads per node [4, 5]. This framework will necessarily lead to systems with a large number of components, with an associated large communication cost for data exchange. This will inevitably lead to simulations that are communication-limited rather than being limited by CPU-time. The presence of many components and the increasing complexity of these systems (e.g. more and smaller transistors, and lower voltages) can become a liability in terms of system faults. It widely accepted, in fact, in the scientific and engineering communities that exascale systems will suffer from errors and faults much more frequently than the current petascale systems. This will

likely make the current parallel programming models and approaches for resiliency to be unsuitable for fault-free simulations across many cores for reasonable amounts of time.

In general, system faults can be grouped under two main categories, namely hard and soft (or silent) faults. Hard faults are usually catastrophic because they cause partial or full computing nodes to fall, or the network to crash. These faults, however, have an evident impact on the run and the system itself. Silent errors, on the other hand, are more subtle because they never get detected, e.g. in the case of bit-flips. The reason is that their effect is simply to alter in some way how the information is stored, transmitted, or handled. 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.

Currently, application checkpoint-restart is the most commonly used tool for fault-tolerance. This approach is in fact straightforward and robust, but it is anticipated that in future extreme scale systems it will not work as well because the time for checkpointing and restarting will exceed the mean time to failure [4, 13, 5]. Moreover, checkpointing can lead to substantial overhead depending on the simulation size [2]. Improving reliability and efficiency for future extreme scale systems is thus becoming increasingly more important. This objective is crucial to achieve in the context of large-scale scientific problems, ranging, e.g., from climate predictions, to nano-engineering, medicine and biology, whose complexity can only be tackled with large computing power and time.

Beside the hardware-oriented engineering efforts aimed at improving these systems, key mathematical challenges arise, involving, e.g., how to make a simulation less sensitive to communication bottlenecks, how to formulate a scientific simulation such that it remains well-defined even in the presence of system faults, and how to rigorously assess the predictive fidelity of extreme-scale scientific simulations in this context. Emerging approaches to fault-tolerance include algorithm-based fault tolerance (ABFT) [2, 7, 10, 6], process-level redundancy [21], and algorithmic error correction code [16]. ABFT is labeled as a non masking approach because algorithms need to integrate ABFT by incorporating some level of redundancy [4]. If an error or a fault occurs, data redundancy allows reconstruction of the missing part of the result. It is increasingly more recognized that new approaches are needed to be incorporated at the algorithm level that account for potential faults, so that the algorithms themselves are made more robust and resilient, without relying exclusively on hardware.

This is the framework in which this paper fits. We present a domain-decomposition preconditioner for the solution of 2D partial differential equations (PDEs) that is resilient to both soft and hard faults. The algorithm is the extension of the 1D version developed in [19] and consists of recasting the original PDE problem as a sampling problem, followed by a resilient data manipulation to achieve the final solution update. One of the main features of the algorithm is that we do not characterize all types of system faults that can occur, but focus solely on the information that a simulation provides. For the implementation, we rely on an asynchronous server-client framework, where one or more servers, which are assumed to be "safe", hold the data, while the clients ask for tasks and execute them. This framework provides resiliency to hard faults. Erroneous subdomain solves (e.g. due to soft faults) appear as corrupted data, which is either rejected if a task fails, or is seamlessly filtered out during the regression stage through a suitable noise model. We explore the effect of three different types of faults: hard faults modeling nodes (or clients) crashing; soft faults occurring during the communication of the tasks between server and clients; and soft faults occurring during task execution. These faults are modeled using a Poisson process defined by a failure rate extracted from literature. We demonstrate the resiliency of the approach for a 2D elliptic PDE, and explore the effect of the faults at various failure rates.

The paper is organized as follows. In § 2 we describe the mathematical formulation; in § 3, we illustrate the actual implementation details; § 4 illustrates how we model and inject faults; § 5 briefly describes the test case adopted; in § 6 we discuss the results, and § 7 presents the conclusions.

2. MATHEMATICAL FORMULATION

This work describes a two-dimensional (2D) Partial Differential Equations (PDEs) solver that is resilient to both soft and hard faults. The algorithm illustrated below is the 2D extension of the 1D solver developed in [19]. For the purpose of this work, we revisit the formulation for a generic 2D elliptic PDE 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 the domain Ω. This approach is not restricted to elliptic PDEs, but its extension to other types of PDEs is outside of the scope of this paper, and will be the subject of a future publication.

Figure 1 shows a high-level schematic of the algorithm's work flow. The starting point involves defining a discretization of the computational domain. In general, the choice of the discretization method is arbitrary, potentially heterogeneous across the domain, e.g. uniform, or non-uniform rectangular grid, or a finite-element triangulation, etc.

After discretizing the computational domain, the second step is the *partitioning* stage, in which the target 2D domain, Ω, is split into a grid of $n_{x_1}^{(s)} \times n_{x_2}^{(s)}$ *overlapping* regions (or subdomains), with $n_{x_k}^{(s)}$ being the number of subdomains along the x_k-th axis. In general, the size of the overlap between neighboring subdomains is an arbitrary parameter that, as shown in [19], can play an important role for nonlinear problems, while having only a minor effect for linear PDEs. 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 $n_{x_1}^{(s)} \times n_{x_2}^{(s)}$ subdomains $\Omega_{ij}^{(s)}$, and their corresponding boundaries $\Gamma_{s_{ij}}$, for $i = 0, \ldots, n_{x_1}^{(s)} - 1$, and $j = 0, \ldots, n_{x_2}^{(s)} - 1$, where $\Gamma_{s_{ij}}$ represents the boundary set of the ij-th subdomain $\Omega_{ij}^{(s)}$.

The algorithm considers each subdomain to be *independent*

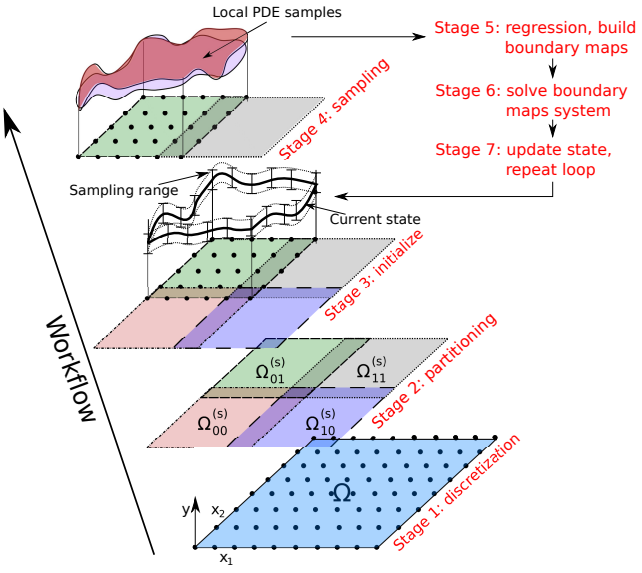

Figure 1: Schematic of the workflow of the algorithm. For clarity, starting with stage 2 we only show the steps for $\Omega_{01}^{(s)}$ but the same "operations" are applied to all subdomains.

of the others. One potential constraint is that, for the purposes of the algorithm, within each subdomain the discretization should allow to easily extract the solution at the locations corresponding to the intersecting boundaries. This can be easily obtained by using a uniform rectangular grid, and suitable domain decomposition, such that the grid points coincide with the boundaries of the intersecting subdomains. This is the case illustrated in figure 1, and also adopted for simplicity in this study. If this was not the case, one would have to rely on interpolation to extract these data. Exploring heterogeneous discretization involving more general discretization methods and their effect on the algorithm is outside of the scope of this work and is being addressed elsewhere.

One of the advantages of the above decomposition for the elliptic problem in Eq. (1) is that if we knew the true solution along the subdomain boundaries, then this information could be used as boundary condition within each subdomain, to perform a single local solve that would then yield the full solution over the full domain, Ω. Consequently, it is sufficient to 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 $i = 0,\dots,n_{x_1}^{(s)}-1$, and $j = 0,\dots,n_{x_2}^{(s)}-1$. Due to the overlapping, each subdomain $\Omega_{ij}^{(s)}$ includes *inner* boundaries, $\Gamma_{s_{ij}}^{in}$, i.e. the parts of the boundaries contained within $\Omega_{ij}^{(s)}$ that belong to the intersecting (neighboring) subdomains. The core of the algorithm, as shown in [19], relies on exploiting the relationship between the solution at the subdomain boundaries as follows: within each subdomain $\Omega_{ij}^{(s)}$, our goal is to find 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, $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}^{in}}$. These maps can be written compactly as

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

for $i = 0,\dots,n_{x_1}^{(s)}-1$, and $j = 0,\dots,n_{x_2}^{(s)}-1$. 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 $\mathbf{y}(\mathbf{x}) = \mathcal{F}\mathbf{y}(\mathbf{x})$, 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. As outlined in [19], even though general (non-)linear solvers can solve the fixed point problem, this approach is not the best because it involves an overhead due to global communication and would require on the fly subdomain solutions to evaluate the maps. The method adopted in [19], which we carry over to the work presented here, is to construct *approximations* (or surrogates) of the boundary-to-boundary maps, which we call $\tilde{f}^{(ij)}$. One of the main advantages of this approach is that the computations can be done *locally* and *independently* within each subdomain without requiring information from the neighbors. This allows us to satisfy data locality and avoid the overhead due to communication, which is crucial to achieve scalability on extreme scale machines. To build these surrogate maps we use a sampling strategy that involves solving the equation locally on each subdomain for sampled values of the boundary conditions on that subdomain. These samples are used within a regression approach to "infer" the approximate boundary maps. One can thus view the construction of these surrogate maps simply as a step in which we "learn" the coefficients of the boundary maps. In general, for non-linear problems, the maps are non-linear and using linear surrogate maps will carry an additional source of discrepancy, due to the linear approximation of a generally non-linear map. For linear PDEs, instead, as shown in [19], the boundary maps are linear as well.

To build these maps we need a current "state" of the solution at the subdomains boundaries, and a sampling range that is used to generate samples within each subdomain, see stage 3 in figure 1. Using the current solution state and the current sampling range values, we can generate samples within each subdomain which are then used in a regression stage to build the approximate maps. This stage plays a key role for addressing soft faults. As shown in [19], in fact, when inferring linear maps, using a suitable ℓ_1-noise model one can seamlessly filter out the effects of few corrupted data. The ℓ_1 noise model allows us to find 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. Following the construction of the surrogate boundary-to-boundary maps, we can then solve 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. For the case of linear PDEs, because the boundary-to-boundary maps are linear and given that no faults occur, the approximate solution obtained after one iteration coincides with the true solution. An important measure of the accuracy of the current solu-

tion $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}}$ is the *residual* vector, defined as

$$\mathbf{z}^{(T)} = \mathcal{F}\mathbf{y}^{(T)} - \mathbf{y}^{(T)}, \tag{3}$$

which can be computed by extra subdomain solves using boundary conditions defined by the current solution $\mathbf{y}^{(T)}$, and subtracting the corresponding current solutions $\mathbf{y}^{(T)}$ from the resulting values at all boundaries. It follows from the definition that the residual (3) vanishes if the current solution $\mathbf{y}^{(T)}$ is the exact solution.

The above outline of the algorithm shows that the original PDE problem is reformulated mainly in terms of a sampling problem, followed by resilient data manipulation to achieve the final solution update. It is important to mention that knowing the boundary-to-boundary maps discussed above gives us the framework to potentially solve the target PDE over the target domain Ω with any boundary conditions. This is because once the maps between the subdomain boundaries are known, then solving for any boundary condition simply translates into doing one solve of the fixed point system for the new boundary condition. In the context of performance analysis, a comparison against existing solvers should not evaluate the cost of a single run. The comparison should be seen in a larger context, namely one involving ensembles of runs for uncertain boundary conditions as well as system faults. This is currently being pursued and will be presented elsewhere.

3. ALGORITHM IMPLEMENTATION

Two separate parallel, C++ implementations of the algorithm have been developed, "distributed" and "task manager" based implementation. The packages are designed using *inheritance*, *polymorphism* and data *encapsulation*, and other core properties of object-oriented programming (OOP), which contribute to well-structured, modular, and portable implementations. These properties greatly facilitate the addition of new functionalities, and the modification of existing capabilities. Both versions rely on the Boost MPI library for the communication, which itself wraps the Message Passing Interface (MPI) library.

The first version is based on mapping one subdomain to one MPI process, such that each MPI process exclusively handles a specific subdomain and all the local information and computations, thus making the package fully "distributed". The alternative is a task manager-based (TM) implementation, which consists of grouping the available MPI processes into servers and clients. The servers are assumed to be safe units holding the data, whereas the clients are designed solely to accept and perform work. A client can be a full computing node or, more generally, simply a set of MPI processes.

Both implementations have advantages and disadvantages, and reveal marked differences. For instance, it is evident that the distributed version minimizes the amount of communication, while the TM involves substantial data exchange between servers and clients. From a resilience standpoint, the distributed version would not be able to tolerate a hard fault, e.g. an MPI process or a full node crashing, because part of the data would be indefinitely lost. On the contrary, the TM is inherently resilient to the scenario of clients crashing, since that translates into only missing tasks as the actual data is still safely held by the server. Since the focus of this work is resilience, we rely on the TM implementation to generate all the results presented below. Figure 2 shows a schematic of our TM structure. The starting point is a set of available MPI ranks, some of which play the role of servers, while some play the role of clients. We then form separate clusters each containing a server and, for resource balancing purposes, the same number of clients. These clusters are designed such that all servers can communicate between each other, while the clients within any cluster are only visible to the server within the same cluster. The data is distributed among the servers, and these are assumed to be highly resilient (safe or under a sandbox model implementation). The sandbox model assumed for the servers can be supported by either hardware or software. The former assumption is supported by hardware designer specifications on the variable levels of resilience that can be allowed within large computer systems. In the case of software support, a sandbox effect can be accomplished by a programing model relying on data redundancy and strategic synchronization [14, 3, 11]. Since the servers hold the data, they are responsible for generating work in the form of tasks, dispatching them to their pool of available clients, as well as receiving and processing completed tasks. Since each client can comprise multiple MPI ranks, when it is ready to perform new work, it is its root process that receives the new task to perform. Once the task is received by that root process, it is then broadcast to all the ranks in the client so that the client as a whole can work in parallel to solve the task. All communications between server and clients are done with non-blocking operations, allowing us to overlap them on the server side with the computational operations involved in the creation and processing of the tasks. Given that our code involves complex C++ objects, e.g. the tasks objects themselves, we leverage the Boost serialization library to enable object communication via MPI. For this study, we concentrate on a single cluster, i.e. a single server with multiple clients, but we anticipate that this configuration is not suitable for large problems and for targeting scalability studies. To this end, one should envision having multiple servers, as shown in figure 2. Since the focus of this work is resilience, we omit scalability stud-

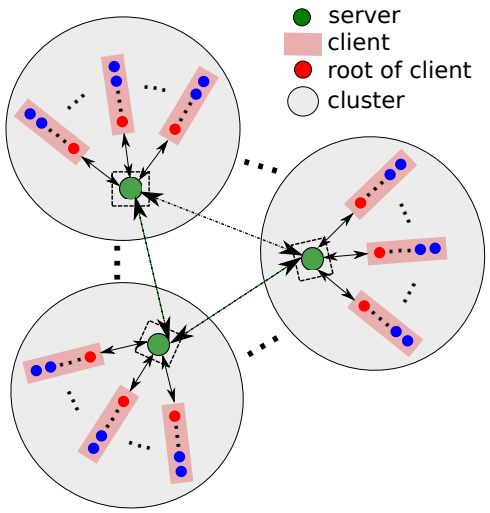

Figure 2: Schematic of our task manager network.

ies and comparison against the distributed version, which will be reported elsewhere. For all the results below, unless stated otherwise, we focus on a scenario where we have one MPI rank that is the server, and 62 clients, each consisting of 2 MPI processes, yielding a total of 125 running MPI processes.

4. SOFT AND HARD FAULTS

Reliably modeling faults in a computing platform is not an easy task. Various attempts have been made to find the statistical distribution that best fits the data extracted for real systems, see e.g. [12, 15, 17, 22, 18, 20] and references therein. Faults can be grouped under two main categories, namely hard and soft (or silent). Hard faults have many causes, and their effects are usually catastrophic because they cause partial or full computing nodes or network failures. These faults, however, can be "seen", in the sense that they have an evident impact on the run and the system itself. Silent errors, on the other hand, are more subtle and can go undetected. The reason is that their effect is not to break a particular system component, but simply alter in some way how the information is stored, transmitted, or handled. The key feature of silent errors is that, being undetected, there is no opportunity for an application to recover from the fault when it occurs. Designing algorithms that are resilient to silent errors is a very important line of future research [5]. If the algorithm itself is designed to include capabilities to seamlessly filter out these faults, then there is no need to detect them.

To test the resiliency of our algorithm to both hard and soft faults, we synthetically inject faults into the system as follows. Hards faults are modeled as entire clients crashing. More specifically, we model this by assuming that if any one of the MPI ranks defining that client dies, then the entire client is deemed as dead. This picture is taken to be consistent with the realistic scenario of a entire node failing. Since MPI does not yet allow ranks within a communicator to fail for real, we cannot actually kill the ranks because the full run would crash, so we simulate that by simply making those ranks sit idle for the rest of the computation. Dead clients stop communicating with the server, and, therefore, do not receive any additional work to do. Silent errors, on the other hand, are modeled as random bit-flips corrupting the data at three possible stages: during the transmission of a task from a server to a client; during the task execution; and, finally, during the transmission of a completed task from a client to its server.

4.1 Failure Distribution

So far we have discussed the type of faults that we model, how we inject them in the algorithm, but we are missing a model to simulate their occurrence. To this end, we assume an exponential failure density function, $f(t)$, which describes the inter-arrival time between events in a Poisson process, in which events occur continuously and independently at a constant average rate. Using an exponential distribution implies that the process is assumed "memoryless", i.e. each event is independent of the other. Of course, if more realistic models were used, the occurrence of the faults would be different, but this would not affect the results of this study as we focus on how our approach handles faults, regardless of how they occur. This justifies our choice of the simplest possible model. The exponential distribution is defined by a single parameter, rate of failure r, and can be written as

$$f(t) = r \exp^{-rt}. \tag{4}$$

Knowing the failure density function, $f(t)$, we can define the failure distribution, $F(t)$, which is simply the cumulative distribution function

$$F(t) = \int_0^t r \exp^{-r\tau} d\tau = 1 - \exp^{-rt}. \tag{5}$$

As previously stated, in this study we model three different scenarios of faults: hard faults, communication soft faults, and computation soft faults. In order to define suitable failure rates for modeling their occurrence, we rely on the data in [20]. We extract failure rates by scaling up the results found in [20] assuming future architectures to have a 10^4-way local concurrency within nodes (stemming from a combination of cores and threads), and comprising 10^5 nodes. Table 1 reports the computed failure rates for each of the fault category target in this work. From the table it is clear that the rates for the communication faults are much larger than the others. This is due to the fact that while the rates for communication faults have been scaled by considering a network connecting 10^5 nodes, the others are based only on the assumption of a 10^4-way local concurrency within nodes. Figure 3 shows a representative example of the failure distribution $F(t)$ for the communication faults for each of the six failure rates computed.

From an implementation standpoint, to simulate the occur-

Table 1: Failure Rates (n_{faults}/sec)

	Hard Faults	Soft Faults Computation	Soft Faults Communication
$r_1 =$	0.00005	0.00004	0.00069
$r_2 =$	0.00009	0.00009	0.00140
$r_3 =$	0.00018	0.00017	0.00270
$r_4 =$	0.00034	0.00035	0.00550
$r_5 =$	0.00072	0.00070	0.01100
$r_6 =$	0.00090	0.00087	0.01400

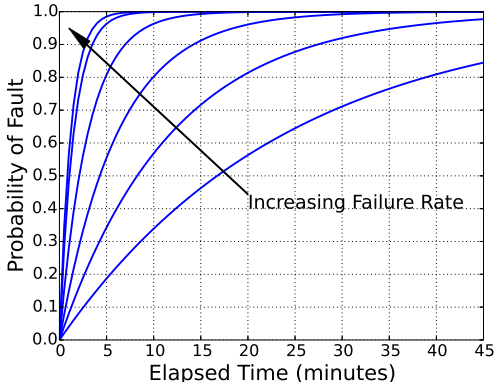

Figure 3: Failure distribution $F(t)$ for the communication soft faults for the six different failure rates explored in this study.

rence of a fault for a target operation we proceed as follows. For a given failure rate, we draw a sample from a standard uniform random number, and extract from the corresponding failure density $F(t)$ the amount of time until the next fault occurs. We then measure the execution time for the target operation to complete, and if that time exceeds the next failure time, then a fault is triggered. Once the fault is triggered, we proceed to simulate the effect of the fault as previously described.

4.2 Handling Faults

The TM implementation handles two kinds of tasks, namely sampling and regression tasks. The sampling stage is designed such that we keep generating tasks until a sufficient number of samples is collected for each subdomain to have a well-posed regression stage. If enough samples are not collected, the problem is under-determined, and we know in advance that the regression would not succeed. If this is the case, then we simply repeat the iteration since running the regression anyway would be a waste of resources. During the sampling stage, if a task fails, it is simply discarded by the server and its data is not used. During the regression, instead, if a task fails, the server tries to rerun it for a fixed number of times (up to 5 times in the cases described in this study), and, eventually, if none of these succeed, the server simply does not use the corresponding data. During this stage, the server knows exactly how many regression tasks need to be executed. Since the regression is a fundamental part of the algorithm for the final fixed point solve, the server keeps track of what tasks come back and, eventually, when all regression tasks have been run at least once, it recreates and executes those regression tasks that were lost. In both cases, if a task is successful, then its data is used, but there is no guarantee that the data is "right", since it could be corrupted data due to the modeling of soft faults. In the present work, when a task is processed, we verify that the data stored in that task is "meaningful", i.e. the data does not contain NaN or Inf. If that is the case, then the task is simply discarded.

5. TEST CASE DEFINITION

As a test case for demonstrating the algorithm, we consider the following 2D steady diffusion equation

$$\frac{\partial}{\partial x_1}\left(k(\mathbf{x})\frac{\partial y(\mathbf{x})}{\partial x_1}\right)+\frac{\partial}{\partial x_2}\left(k(\mathbf{x})\frac{\partial y(\mathbf{x})}{\partial x_2}\right)=g(\mathbf{x}), \qquad (6)$$

where $\mathbf{x}=\{x_1,x_2\}$, the field variable is $y(x_1,x_2)$, $k(x_1,x_2)$ is the variable 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, and the following diffusivity and source:

$$k(x_1,x_2)=0.5*(9.0+9.0*\tanh\left(\frac{d(x_1,x_2)}{0.01}\right)+1.0, \quad (7)$$

$$g(x_1,x_2)=0.5*(2.0+2.0*\tanh\left(\frac{d(x_1,x_2)}{0.01}\right)-1.0, \quad (8)$$

where $d(x_1,x_2)=0.25-\sqrt{(x_1-0.5)^2+(x_2-0.5)^2}$. This yields a non-trivial solution due to the steep gradient in the diffusivity and the source term, which can pose some challenges in the numerical solution if the spatial discretization is not sufficiently fine.

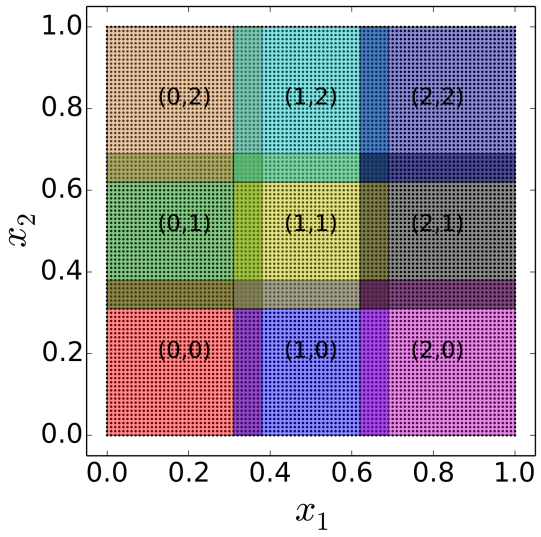

Figure 4: Schematic showing the $n_{x_1}^{(s)}=n_{x_2}^{(s)}=3$ subdomains partitioning, as well as the underlying $n_x=n_y=101$ discretization grid

To test our algorithm, we adopt a $n_{x_1}^{(s)}=n_{x_2}^{(s)}=3$ partitioning, with an underlying global mesh of $n_x=n_y=101$ and a local subdomain grid size of about 37^2. Hereafter, we refer to this problem using the label $S9g_s37^2$, where $S9$ stands for 9 total subdomains, and 37^2 is the reference local grid within each subdomain. The overlap between adjacent subdomains is set to 7 grid cells along both x_1 and x_2. A representative plot of the discretization grid, the resulting decomposition, the subdomains and their overlapping is shown in figure 4. Unless stated otherwise, this is the test case used to run the results below. To numerically solve the PDE within each subdomain during the sampling stage, we rely on a second-order finite difference (FD) approximation over the local rectangular mesh. Due to linearity, the FD approximation yields a linear system of equations, which is solved using the AztecOO package in Trilinos which provides parallel solvers for large linear systems. As previously stated, our algorithm is independent of the type of solver used within each subdomain. We remark that for the purpose of this work the overlap size is not a critical parameter because for linear problems, like the one adopted here as test case, it does not affect the convergence. It can have a substantial impact if the target PDE was non-linear, but this analysis will be the subject of a separate publication.

6. RESULTS

6.1 No Faults

For demonstration purposes we first run the algorithm without any faults. Due to the problem's linearity, the maps between the subdomain boundaries are linear, so the solution is obtained after a single iteration. For our algorithm, the solution is obtained along all the boundaries of the subdomains, but not on the inner grid points. Figure 5 shows the surface plot of the precomputed solution (grayscale), superimposed to the solution along the boundaries of all subdomains. Knowing the state at the boundaries of all the subdomains fully defines the solution, since we are dealing

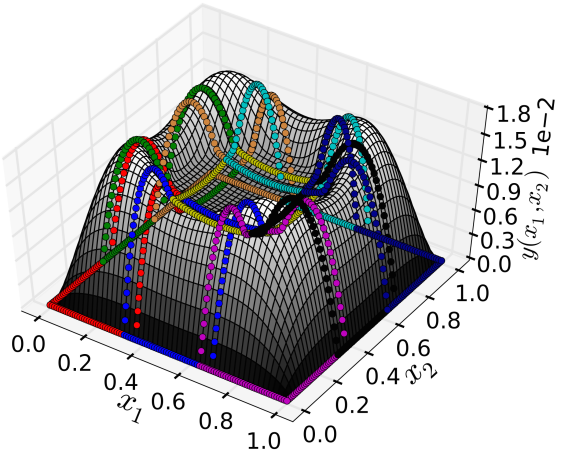

Figure 5: Representative solution of the 2D linear diffusion equation superimposed to solution at the boundaries of the subdomains obtained for a sample no-faults run. The colors used to plot the solution at the boundaries match the ones used in figure 4.

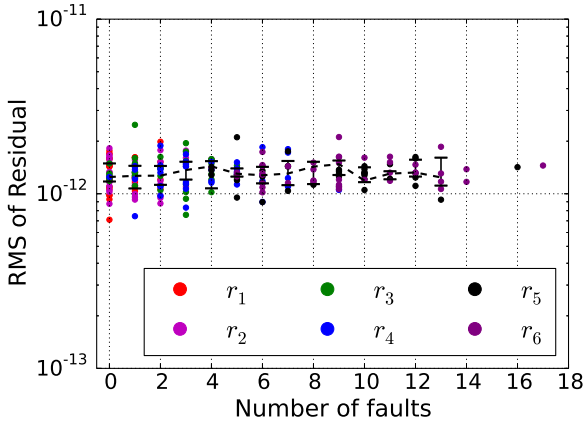

Figure 6: Root-mean-square value of the final residual plotted as a function of the number of *hard* faults. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles. The markers are color-coded based on the failure rate they belong to.

with an elliptic PDE. To find the solution over the inner grids of the subdomains, we would need an additional step in which we use the boundaries state as boundary conditions to perform one more single PDE solve over each subdomain. This would yield the full solution over all grids points in the mesh. For brevity, this step is omitted here, because it is sufficient to know the solution at the subdomains' boundaries.

The first part of the results focuses on exploring the effects of each type of fault individually, and what is the role of the failure rate. Given the "randomness" associated to both the algorithm (i.e. during sampling stage) and the faults occurrence, we explore the effects of the different failure rates in terms of ensembles of runs. To this end, for each value of the failure rate shown in Table 1, we run $N = 40$ simulations for the $S9g_s37^2$ case. To explore the effects of problem size, on these results, a smaller ensemble $N = 10$ was ran for a bigger problem. This larger case is referred to as $S4g_s103^2$ and involves an underlying grid of $n_x = n_y = 201$, partitioning of $n_{x_1}^{(s)} = n_{x_2}^{(s)} = 2$ subdomains, and a local subdomain grid size of 103^2.

6.2 Hard Faults

The effect of hard faults on the algorithm implementation are first explored using the $S9g_s37^2$ test case. We first focus on the convergence results. Figure 6 shows the dependence of the root-mean-square (RMS) of the final residual as a function of the number of faults for all runs in the $S9g_s37^2$ test case. The data points are color-coded based on the failure rate they belong to. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles. This figure is significant because it proves the resiliency of our algorithm with respect to hard faults, since all the runs converge, with no runs failing, regardless of the number of hard faults that hit the "system".

We now explore how the faults affect the run time. Figure 7 shows two sets of data: in black, we plot the number of hard faults for all the ensemble runs as a function of the failure rate, and superimpose the error bars displaying the 0.25 and 0.75 quantiles, as well as the trend for the 0.5 quantile (dashed-line); in blue, we show the fraction of runs that completed without faults as a function of the failure rate. The plot reveals that as the failure rate increase, the number of faults occurring increases, on average, monotonically, ranging from zero for the smallest rate, r_1, to a maximum value of 17 for r_6. We also note that even though we have an ensemble of 40 runs for any given value of the failure rate, the actual variance of the number of faults is not large. This is because several runs have the same number of faults, even though these faults might affect different clients. For instance, for the lowest rate r_1, we only observe 0, 1 or 2 faults, whereas for the largest rate r_6, we observed a minimum of 5 and a maximum of 17 faults. Of course, this is an effect of the finite-time of the simulation, because if we were to run these cases long enough, eventually all clients would fail. The other data set plotted in the figure shows the fraction of runs that complete without encountering a fault. The data reveals that even for the lowest rate case, only about 60% of the runs do not have faults. This value drops to zero for r_5 and r_6.

For illustration purposes, figure 8 shows a snapshot of the status of the clients for one representative run for each failure rate. The plot can be interpreted as follows. The 62 clients cl_i, for $i = 0, \ldots, 61$, that we have available are placed along the angular coordinate; the radial coordinate identifies the failure rate, and increases as we move outward. It follows that each marker in the plot represents the status of a client during a representative run extracted for a given failure rate. Red markers identify clients that have failed at some point during the run, while blue markers represent clients that are alive. For this particular run, we can see that for the smallest failure rate, r_1, all clients remain alive. On the contrary, for the largest case we can see that 9 clients are

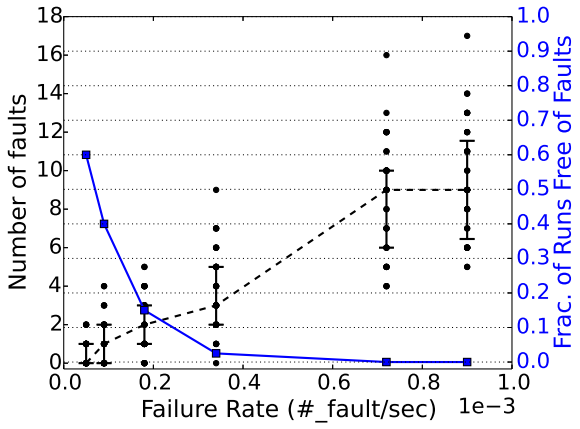

Figure 7: Number of faults for all ensemble runs (black circles) plotted as a function of the failure rate obtained for the *hard* fault case. The errors bars are obtained for the 0.25 and 0.75 quantiles, while the dashed line connects the 0.5 quantiles. The dataset in blue shows the fraction of runs that complete without faults. All results are obtained for the $S9g_s37^2$ case.

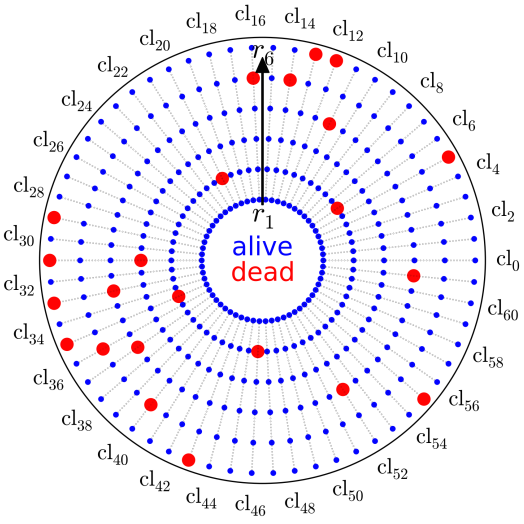

Figure 8: Status of all 62 clients (distributed along the angular direction) for a representative simulation of each of the failure rates r_i, $i = 1, \ldots, 6$. A red marker identifies a client that has died during that run, while a blue marker identifies a client that is alive.

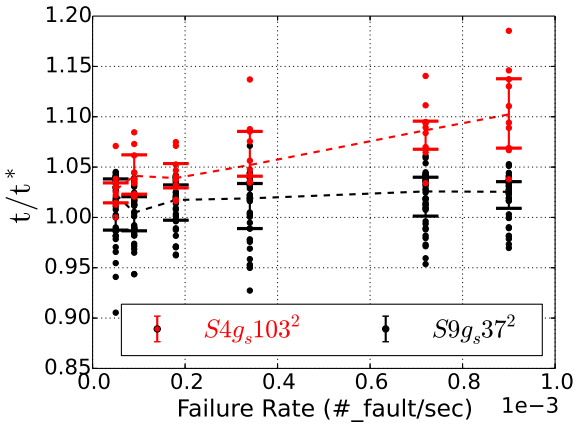

Figure 9: Time per iterations t normalized by t^*, which is the corresponding time for the no faults case, plotted as a function of the number of *hard* faults. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles.

dead. This is just a representative picture, and it would change if a different run was selected.

To investigate what is the effect of the faults on the runtime, figure 9 shows for each run of each rate, the time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of the failure rate. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles. We remark that in all these hard fault cases, the solution is obtained after a single iteration, which means that the time per iteration reported in figure 9 is also the time to converge. The data reported in black shows the results obtained for the reference test case $S9g_s37^2$. While the data displayed in red, shows the results obtained for a bigger problem size, test case $S4g_s103^2$. The two data sets reveal different trends: the data set obtained for the smaller case, $S9g_s37^2$ (shown in black) shows a slowly growing trend as the number of faults increases, but the noise is too large and, thus, the trend is not clear. Overall, the overhead runtime that we face for the runs with faults seems to only weakly depend on the number of faults. Even though at first glance this result might be surprising, it is the consequence of dealing with a problem that is too small in terms of computational load. What the data reveals is that the tasks that are generated for the current problem are not sufficiently intensive such that even when we lose 16 clients out of 62, we do not see much impact on the overall execution time because we are bounded by the communication cost. This explanation is supported by the data plotted in red, which was obtained for the larger problem, $S4g_s103^2$ involving finer grid. The increasing trend is clearly visible for the $S4g_s103^2$ case. By making the problem bigger and, thus, more computationally intensive, the computational cost surfaces, yielding a net growing trend in the runtime cost when some of the clients are killed.

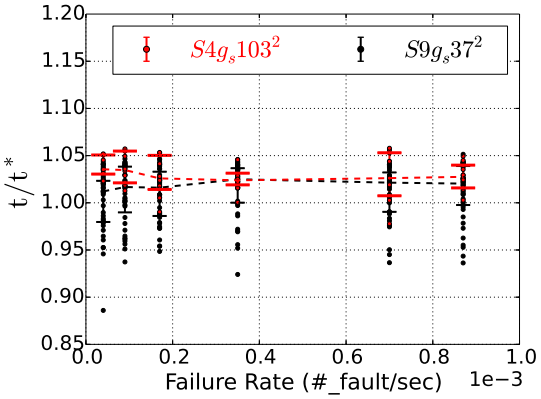

Figure 10: Time per iterations t normalized by t^*, which is the corresponding time for the no faults case, plotted as a function of the number of *computation faults*. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles.

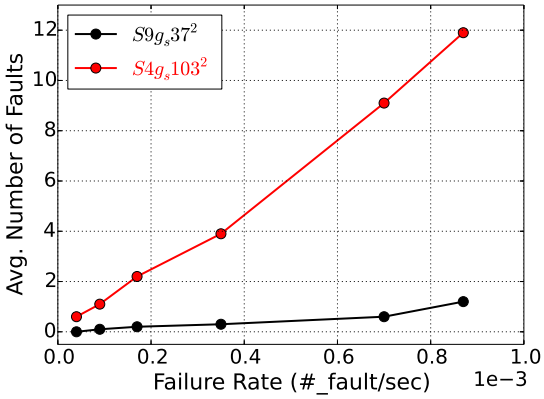

Figure 11: Average number of faults computed over the ensemble runs as a function of the failure rate obtained for the *computation faults*.

6.3 Soft Faults During Computation

While the effect of a hard fault effectively translates into missing data, soft faults occurring in the clients during the task execution can have different consequences. In this case, if a soft fault occurs it can either cause the task being executed to fail, which would be detected by a convergence check on the subdomain, or if the fault results in a very small perturbation, then some iterative solvers or regression algorithms will reach a solution. We anticipate that in the present work, we do not observe the latter scenario. Tasks that are affected by computational soft faults are not lost, and depending on whether it is a sampling or a regression task, we have two different scenarios. For a sampling task, the server receives it back from a client, and if it is a failed task, it is discarded, if not, it is processed. For a regression task, instead, if the task failed, then the server tries to rerun the same task up to 5 times. If the number of attempts reaches the maximum value, the server simply disregards the task so the data stored in that task is not used. This implies that the server does not update the maps coefficients for the boundary for which that task was constructed.

Figure 10 shows for each run of each rate, the time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of the failure rate. The data reported in black, once again, shows the results obtained for the reference case $S9g_s37^2$. The data displayed in red, shows the results obtained for a bigger problem, case $S4g_s103^2$ which runs for a smaller ensembles ($N = 10$) for all failure rates. The errors bars correspond to the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles. The results indicate that the soft faults occurring during the computational stage of the clients do not strongly affect the time per iteration. In other words, for the $S9g_s37^2$ case, the overhead for completing the runs with respect to the no fault case due the presence of computation soft faults is very weakly dependent on the failure rate. The trend does not change if we consider the larger problem ($S4g_s103^2$) with results shown in red. In this case, even though the tasks themselves are more expensive to run, the overall trend is the same and the results do not depart too much from the $S9g_s37^2$ case. This suggests that for the current settings, the combination of number of faults occurring and computational load is not large enough to have an effect as strong as we have seen for the hard fault case. We can thus draw the conclusion that losing computing nodes or clients has a greater impact on the runtime.

Figure 11 shows the average number of faults computed over the ensemble runs as a function of the failure rate obtained for the case with a 3×3 subdomains partitioning and an underlying grid of 101^2 (case $S9g_s37^2$ shown in black), and for the case with a 2×2 subdomains and underlying grid of 201^2 (case $S4g_s103^2$ shown in red). For this type of fault, we observe limited number of faults occurring, across all runs. More specifically, for a fixed value of the failure rate, the number of faults increases with the problem size. The reason behind these results is that if the task execution is completed quickly enough, then it is less likely that a fault occurs. As the problem becomes larger and larger, tasks become more and more expensive to run, and, thus, can be hit more frequently by faults. In both cases, however, the number of faults hitting the clients is still too small to have any substantial impact on the runtime, as shown by figure 10. Also for this type of faults, all the runs converge successfully, and the RMS of the residual for this category of faults behaves similarly to the one shown for hard faults, further confirming the resiliency of the algorithm. For brevity, this plot is omitted.

6.4 Soft Faults During Communication

In this subsection we explore the effects of the fault occurring during the communication operations between server and clients. This type of fault is the most complex for resilience purposes, because it involves undetectable silent errors that corrupt the data in the tasks objects. A suitable example is one where we have a task object that is being sent from a server to a client, and there is some memory corruption due to cosmic rays that corrupts the network. This would affect the actual data owned by the object, or

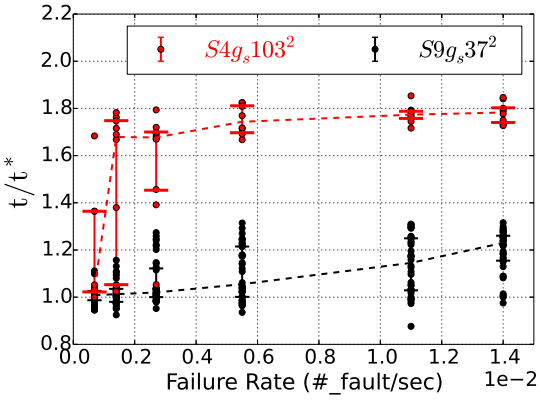

Figure 12: Time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of failure rate for *communication faults*. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles.

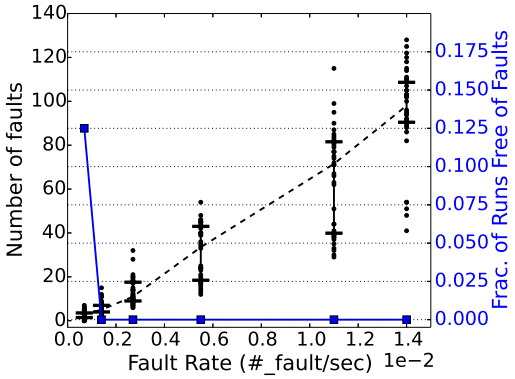

Figure 13: Number of faults for all ensemble runs (black circles) plotted as a function of the failure rate for *communication faults*. The errors bars are obtained for the 0.25 and 0.75 quantiles, while the dashed line connects the 0.5 quantiles. The dataset in blue shows the fraction of runs that complete without faults.

even the object itself being completely corrupted. This effect is what we are trying to model with this type of fault. To this end, our approach relies on the fact that if a fault is triggered, then we synthetically corrupt all the data of that object. There is an important distinction to make between messages that are traveling from the server to a client, and those that are traveling from a client to a server. In the first case, when the server sends a task to a client and a fault occurs, the client receives that task object carrying corrupted data. We remark once again that these are silent errors so there is no way for the client to know that the data is corrupted, unless the data contains NaN or Inf. After receiving the task, the client proceeds with its execution. If the data is corrupted, and depending on the size of the corruption, that task is likely to fail. If that is the case, as mentioned in the previous section the server will then decide what to do next, i.e. whether to rerun that task or discard it. The situation is different if the fault occurs when a task is being transmitted from a client to a server. In this case, the fault most likely corrupts the data owned by that task, but the server will use that data to do the update of the local solution. Again, this is because the server does not have any way to know that the data coming in is corrupted. Even if a subset of the samples data is corrupted, the ℓ_1 regression is capable to filter out these effects and find the right solution. This procedure does not always work as well, because we might have too many corrupted data, or the magnitude of the perturbation is so large that the solver cannot overcome. If this is the case, then it is likely that using these data will lead to a bad solution, which would imply that the algorithm launches another iteration since the expected convergence is not achieved. If many faults occur, then this can cause the run to perform multiple iterations before converging. Figure 12 shows the time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of failure rate. The results are shown for all the runs of the $S9g_s37^2$ case (black curve). The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles. The figure shows that faults occurring during communication play a key role, since they cause a net increase in the computational cost of a single iteration. More specifically, we can see that the cost increases by 20% for the largest failure rate, which is not a negligible amount also considering that these results are obtained for the small problem, $S9g_s37^2$. The dataset plotted in red shows results obtained for $S4g_s103^2$. In this case, we note that the runtime overhead per iteration with respect to the no fault case increases rapidly for smaller rates, to eventually settling into a plateau. The data also reveals that this larger problem $S4g_s103^2$ is more affected by soft faults than the smaller case. This is because the number of faults occurring for this problem is larger. This is due to the fact that task objects for the bigger case are more expensive to exchange via MPI, implying that these communication operations take longer, and are thus more susceptible to be hit by a fault. For brevity, we omitted the plot for the convergence, but we remark that in all cases, like shown for the other types of faults, the runs complete successfully, but need more than a single iteration. Figure 13 shows two sets of data: in black, we plot the number of faults for all the ensemble runs in case $S9g_s37^2$ as a function of the failure rate, and superimpose the error bars displaying the 0.25 and 0.75 quantiles, as well as the trend for the 0.5 quantile (dashed-line); in blue, we show the fraction of runs that completed without faults as a function of the failure rate. The total number of faults that hit the run is very large in this case, ranging from zero for r_1, to more than 130 for the r_6. This is because the rates used for this type of fault are quite large, and, thus, it is more likely that faults occur. It is interesting to see that only for the smallest rate, r_1, we have some runs that complete without encountering any fault. In all other cases, all the runs have at least one fault.

6.5 Mixed Faults

The results discussed above explored each fault category individually. This analysis allowed us to extract some patterns and highlight what the effects each type of fault has on the

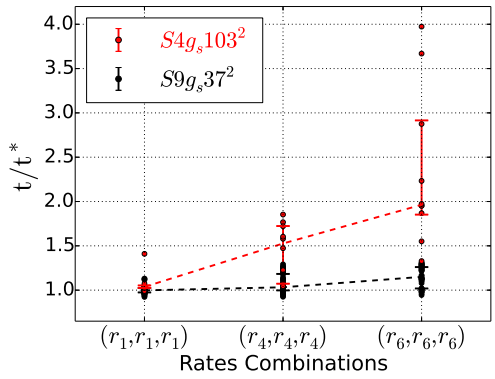

Figure 14: Time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of the mixed fault case.

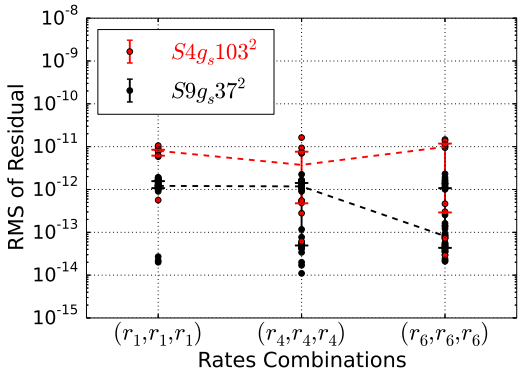

Figure 15: RMS of the residual as a function of the mixed fault case.

runs. This scenario, however, is not a real one because in a real system we cannot turn off target faults arbitrarily. It is thus important to explore how the algorithm behaves when all faults are activated and can occur concurrently. To this end, we choose three different scenarios, one involving a case where all three faults are described by the smallest value of the failure rate r_1; a second case involves all three faults with rate r_4; finally, as last case, we have r_6 for all three types. Figure 14 shows the time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of failure rate. The results are shown for both the small and large problem ($S9g_s37^2$ and $S4g_s103^2$ respectively), with errors bars obtained for the 0.25 and 0.75 quantiles, and the dashed line connecting the 0.5 quantiles. As expected, the plot reveals that the overhead cost per iteration due to the presence of faults increases monotonically as a function of the failure rate for both problems considered. On the one hand, for the small problem ($S9g_s37^2$), the overhead cost of one iteration in the presence of faults for the largest faults rate is on average about 20%. This cost substantially increases to 100% for the larger case, $S4g_s103^2$. For a fixed problem, the plot reveals that the gap between the 0.25 and 0.75 quantiles of the data increases proportionally to the failure rate. This is due to the fact that as the failure rate increases, so does the variability in how the faults occur during the runs. From a different viewpoint, for a fixed value of the failure rate, we can see that variability in the data increases with the size of the problem.

We finalize this section showing the resiliency results. To this end, figure 15 shows the RMS of the residual as a function of the mixed fault cases. The plot reveals that all the runs converge. Due to the presence of faults, we observe high variability in data especially at higher values of the rates.

7. CONCLUSIONS

In this study, we presented a PDE preconditioner that is resilient to hard and soft faults, and showed its application to a 2D elliptic problem involving a steady diffusion equation with variable coefficients. The algorithm exploits a novel reformulation of the problem that allows us to cast it into a sampling problem over a set of subdomains such that "data" is generated, and then suitably manipulated to yield the final updating of the solution state. We rely on an asynchronous server-client framework, where one or more servers, which are assumed to be "safe", hold the data, while the clients ask for tasks and execute them. This framework provides resiliency to hard faults since clients that have crashed are ignored and the remaining clients handle all the tasks. Faults occurrence is modeled using a Poisson process defined by a failure rate, and fault types are grouped under three main categories: hard faults, which mimic clients (or nodes) crashing; soft faults during computation, which mimic silent errors affecting the system during the computational work; and soft faults affecting the MPI communication, mimicing the silent errors that can occur within the network when data is being transmitted. Each fault is first explored individually to extract patterns and understand more deeply its impact on the algorithm execution. First, we remark that in all cases, the algoritm always reaches convergence, demonstrating its resiliency. The effect of the faults is to increase the time per iteration and/or the number of iterations that the algorithm needs to run to complete. We showed that hard faults have a substantial impact on the runtime for the larger problem investigated, while only having a minor impact for the small problem. We explained this apparent discrepancy in terms of communcation and compuation cost. For computational faults, we saw that both problems are sufficiently small so that the tasks executions proceed rapidly, and only few faults occur. For the nextwork faults, we saw the effect of the faults to be quite substantial due to the large number of faults happening, as well as the abrubt and undetectable consequences that these faults have. Finally, we showed the result for a more realistic case where all faults can be triggered together. As intutively expected, this scenario is the most dramatic one, leading to runtimes up to four times as big with respect to the no fault case. The analysis presented above is built using a small part of the data available from the simulations explored. A lot of information can be extracted from these runs. Interesting questions concerns the behavior of the algorithm for much larger problems, the scalability of the algorithm, and potential approaches for dimensionality reduction in 2D. These are currently the subject of parallel studies.

8. 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 Number 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.

9. ADDITIONAL AUTHORS

Additional authors: O. LeMaitre (Duke University, email: `olivier.le.maitre@duke.edu`) and H.N. Najm (Sandia National Laboratories, email: `hnajm@sandia.gov` and O.M. Knio (Duke University, email: `omar.knio@duke.edu`)

10. REFERENCES

[1] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32, Piscataway, NJ, USA, 2014. IEEE Press.

[2] 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.

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

[4] 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.

[5] 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.

[6] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM.

[7] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen. Matrix multiplication on gpus with on-line fault tolerance. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 311–317, 2011.

[8] DOE-ASCR. Exascale programming challenges. Technical report, July 2011.

[9] DOE-ASCR. Top ten exascale research challenges. Technical report, Feb. 2014.

[10] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 225–234, New York, NY, USA, 2012. ACM.

[11] 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.

[12] J. Gray. Why do computers stop and what can be done about it?, 1985.

[13] 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.

[14] 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.

[15] T.-T. Lin and D. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, Oct 1990.

[16] K. Malkowski, P. Raghavan, and M. Kandemir. Analyzing the soft error resilience of linear solvers on multicore multiprocessors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[17] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[18] R. Sahoo, M. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Dependable Systems and Networks, 2004 International Conference on*, pages 772–781, June 2004.

[19] K. Sargsyan, F. Rizzi, P. Mycek, C. Safta, K. Morris, H. Najm, O. LeMaitre, O. Knio, and B. Debusschere. Fault resilient probabilistic preconditioner method for one-dimensional pdes. *in preparation*, 2014.

[20] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337–350, Oct 2010.

[21] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 297–306, 2007.

[22] N. H. Vaidya. A case for two-level distributed recovery schemes. In *In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.