Journal Title
XX(X):1–13

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Partial Differential Equations Preconditioner Resilient to Soft and Hard Faults

F. Rizzi[1] and K. Morris[1] and K. Sargsyan[1] and P. Mycek[2] and C. Safta[1] and O. LeMaitre[2] and O. Knio[2] and B. Debusschere[1]

Abstract

We present a domain-decomposition-based preconditioner for the solution of partial differential equations (PDEs) that is resilient to both soft and hard faults. The algorithm reformulates the PDE as a sampling problem, followed by a solution update through data manipulation that is resilient to both soft and hard faults. This reformulation allows us to recast the problem as a set of independent tasks, and exploit data locality to reduce global communication. We discuss two different parallel implementations: a) a single program multiple data (SPMD) version based on a one-to-one mapping between subdomain and MPI processes responsible for both state and computation; and b) an asynchronous server-client implementation where all state information is held by the servers, and clients are designed solely as computational units. We present a scalability comparison of both implementations under nominal conditions, showing efficiency within $\sim 80\%$ for up to 12k cores. We present a resilience analysis under different fault scenarios based on the server-client implementation. This framework provides resiliency to hard faults such that if a client crashes, it stops asking for work, and the servers simply distribute the work among all the other clients 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. We demonstrate the resiliency of the approach for a 2D elliptic PDE, and explore the effect of the faults at various failure rates.

Keywords

Resilience, Scientific computing, Supercomputers, Parallel programming, Distributed computing, Client-server systems, High performance computing, Parallel algorithms, Software engineering, Fault tolerance, Message passing, Fault tolerant systems, Partial differential equations

1 Introduction

Past changes in computer architectures have yielded changes in both hardware and computational models. Exascale simulations are expected to continue along this path, and due to the fast-approaching limit of Moore's law, the change we are about to face might be even more radical than before. Two main features are arising as the defining mark of exascale simulations, namely concurrency and resiliency. Exascale simulations are indeed expected to rely on thousands of nodes, and take advantage of local concurrency through cores and threads per node, as well as accelerators (1; 2; 3; 4; 5). This framework will lead to systems with a large number of components with an associated large communication cost for data exchange. One of the main challenges thus involves reducing the cost of communication. The presence of many components and the increasing complexity of these systems (e.g. more and smaller transistors, lower voltages, and heterogeneous hardware) can become a liability in terms of system faults. Exascale systems will suffer from errors and faults much more frequently than the current petascale systems (4). 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.

Scientists, engineers and software developers face the challenge to address new questions involving, e.g., how to make a simulation less sensitive to communication bottlenecks, how to formulate a scientific simulation to remain well-defined even in the presence of system faults, how to design and develop codes that are reliable and use extensive exception handlers, and how to rigorously assess the predictive fidelity of extreme-scale scientific simulations in this context. Possible approaches to fault-tolerance include algorithm-based fault tolerance (ABFT) (6; 7; 8; 9; 10), process-level redundancy (11), algorithmic error correction code (12) and checkpoint/restart (13). 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 and/or computational units, see, e.g., (10) for an example in the context of PDE solvers, and (14) for fault-tolerant iterative methods. It is increasingly

[1]Sandia National Laboratories, CA, USA
[2]Duke University, NC, USA

Corresponding author:
Francesco Rizzi, Sandia National Laboratories, CA, USA
Email: fnrizzi@sandia.gov

more recognized that new approaches are needed to be incorporated at the algorithm level to account for potential faults, so that the algorithms themselves are made more robust and resilient, without relying exclusively on hardware (5).

This paper presents a domain-decomposition preconditioner for the solution of 2D partial differential equations (PDEs) that is resilient to both soft and hard faults. The work presented here complements and extends the one presented in (15). The algorithm 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. We discuss two different parallel implementations: a) a single program multiple data (SPMD) version based on a one-to-one mapping between subdomain and MPI processes responsible for both state and computation; and b) an asynchronous server-client implementation where all state information is held by the servers, and clients are designed solely as computational units.

We discuss resiliency for the server-client implementation. 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. The occurrence of these faults is modeled as 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. To frame this paper in the proper context, our approach is intended for future exascale platforms, assuming that fault rates will be sufficiently high that checkpoint-restart will not be a feasible option, and current solvers will fail to adequately scale due to system size.

The paper is organized as follows. In § 2 we describe the mathematical formulation; in § 3, we illustrate the actual implementation details; § 4 describes the PDE used in both scalability and resiliency analysis. Scalability results are shown in § 5; the resilience setup and fault model is described in § 6 and the corresponding results in § 7, and § 8 presents the conclusions.

2 Mathematical Formulation

The algorithm illustrated below extends the 1D solver developed in (16). We present below the formulation for a generic 2D linear elliptic PDE of the form

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

where $\mathcal{L}$ is a linear 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. The extension to other types of PDEs is outside of the scope of this paper, and will be the subject of a future publication.

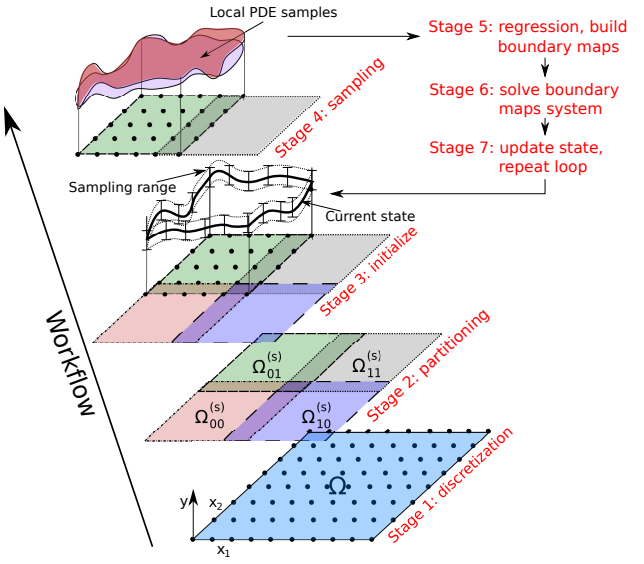

Figure 1. Schematic of the algorithm's workflow. For clarity, starting with stage 2 we only show the steps for $\Omega_{01}^{(s)}$ but the same "operations" are applied to all subdomains.

Figure 1 shows a high-level schematic of the algorithm's workflow. 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.

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. The size of the overlap between neighboring subdomains is an arbitrary parameter playing an important role for non-linear problems, while having only a minor effect for linear PDEs. The effect of the overlap is discussed in (16). 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 $\Omega_{ij}^{(s)}$.

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 yielding 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 $i = 0, \ldots, n_{x_1}^{(s)} - 1$, and $j = 0, \ldots, n_{x_2}^{(s)} - 1$. 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 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 on the inner boundaries, $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}^{in}}$, to the solution on the subdomain boundaries, $y(\mathbf{x})|_{\mathbf{x}\in\Gamma_{s_{ij}}}$. These maps can be written compactly as

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

for $i = 0,\ldots,n_{x_1}^{(s)}-1$, and $j = 0,\ldots,n_{x_2}^{(s)}-1$. 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 discussed in (16), 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 main idea 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. 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, however, the boundary maps are linear as well (16).

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 (16), in fact, when inferring linear maps, using a suitable ℓ_1-noise model one can seamlessly filter out the effects of a 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 the effect of faults is filtered out in the regression stage, the approximate solution obtained after one iteration coincides with 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* vector, defined as $\mathbf{z}^{(T)} = \mathcal{F}\mathbf{y}^{(T)} - \mathbf{y}^{(T)}$ 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 vanishes if the current solution $\mathbf{y}^{(T)}$ is the exact solution.

3 Algorithm Implementation

We have developed two algorithms implemented in C++: one based on the single program multiple data (SPMD), the other on a server-client (SC) programming model. Both versions use the Boost MPI library for the communication, which itself wraps the Message Passing Interface (MPI) library.

The SPMD implementation follows the most common model used in HPC applications. It involves a one-to-one mapping between subdomains and MPI processes, implying that each MPI process exclusively handles a specific subdomain and all its local information and related computations. There are two potential alternatives to this scenario. One would be to map one subdomain onto multiple MPI ranks, thus yielding both data and computation effectively split among multiple ranks. This would be advantageous for problems with subdomains sufficiently large in terms of computational requirement so that having the local computations performed in parallel among multiple ranks can be more efficient. The opposite scenario is one where each MPI rank owns multiple subdomains. This is suitable if the subdomains are small enough in terms of memory needs, and all the local computational workload can be efficiently handled through multi-threading within a single rank.

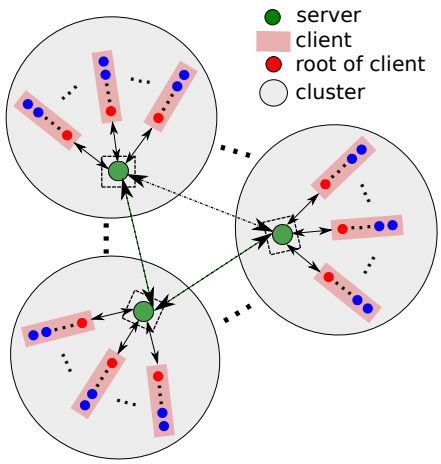

Figure 2. Schematic of our server-client model.

The SC version involves grouping a set of available MPI processes into servers and clients. The servers are safe units holding the data, whereas the clients are designed solely to accept and perform work. Figure 2 shows a schematic of our SC structure. Starting from a set of available MPI ranks, some play the role of servers, while some play the role of clients. Separate clusters are built, 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 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 programming model relying on data redundancy and strategic synchronization (17; 18; 19).

The servers hold the data, generate work in the form of tasks, asynchronously dispatch them to their pool of available clients, as well as receive and process 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. This paradigm can be exploited in certain hardware configurations, because leveraging local communication within a client is more efficient than having the server communicate a task to all the MPI ranks in a client. One example is the case where a client occupies a single node, so that one can exploit in-node parallelism and faster memory access, e.g. in massive multi-core chips. 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.

4 Test PDE

As a test case for all results presented henceforth, we consider the following 2D steady diffusion equation

$$\frac{\partial}{\partial x_1}\left(k(\mathbf{x})\frac{\partial y(\mathbf{x})}{\partial x_1}\right)+\frac{\partial}{\partial x_2}\left(k(\mathbf{x})\frac{\partial y(\mathbf{x})}{\partial x_2}\right)=g(\mathbf{x}), \quad (3)$$

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:

$$k(x_1,x_2)=5.5+4.5*\tanh\left(\frac{d(x_1,x_2)}{0.01}\right), \quad (4)$$

$$g(x_1,x_2)=\tanh\left(\frac{d(x_1,x_2)}{0.01}\right), \quad (5)$$

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.

For demonstration purposes, Figure 3 shows the surface plot of the precomputed solution (grayscale), superimposed to the solution along the boundaries of a 3x3 decomposition. Knowing the state at the boundaries of all the subdomains fully defines the solution, since we are dealing 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.

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. The setup procedure is somewhat arbitrary, in the sense that one does not necessarily have to rely on a pre-defined discretization mesh to choose the subdomain partitioning. We envision, in fact, the case

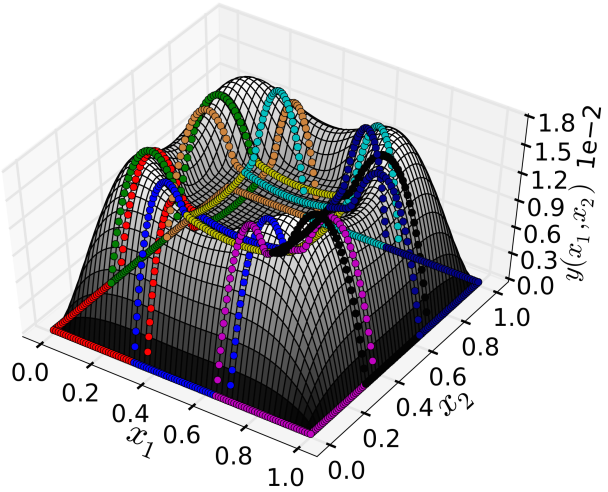

Figure 3. Representative solution of the 2D linear diffusion equation superimposed to solution at the boundaries of the subdomains obtained for a sample no-faults run. The colors used to plot the solution at the boundaries match the ones used in figure 6.

where the user can decide the partitioning first, and then define local discretization within each subdomain, which do not necessarily need to agree. As previously stated, our algorithm is independent of the type of solver used within each subdomain.

5 Nominal Scalability

This section discusses scalability tests for both implementations performed at NERSC on Edison a Cray XC30, with Peak performance of 2.57 Petaflops, Cray Aries high-speed interconnect with Dragonfly topology with approximately 8GB/sec MPI bandwidth. Table 1 lists the parameters used for the scalability runs. In this section, we concentrate on simulations completed in the absence of faults.

SPMD Weak Scaling: Figure 4 shows weak-scaling results obtained for the SPMD case focusing on the two most time consuming stages, namely sampling and regression. Both sampling and regression show efficiency within 90% over the range of cores explored. This was expected for the current setting with one subdomain per MPI rank because these two stages do not involve any communication among ranks, since all information is local and computation within

Table 1. Details of the scalability runs for SPMD and Server-Client implementations.

	SPMD	Server-Client
Subdomains	12^2, 24^2, 48^2, 96^2	6^2, 12^2, 18^2, 24^2
Total Cores	144, 576, 2304, 9216	772, 3088, 6948, 12352
Subdomain size	$\sim 100^2$	$\sim 120^2$
Servers	NA	4, 16, 36, 64
Clients per server	NA	48
Size of each client	NA	4

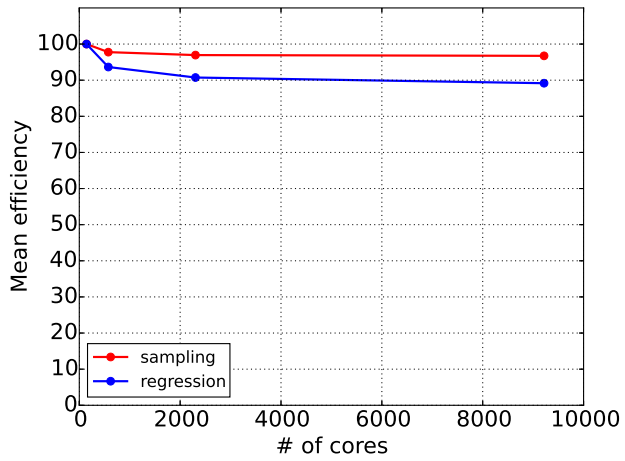

Figure 4. Weak scaling for the SPMD implementation.

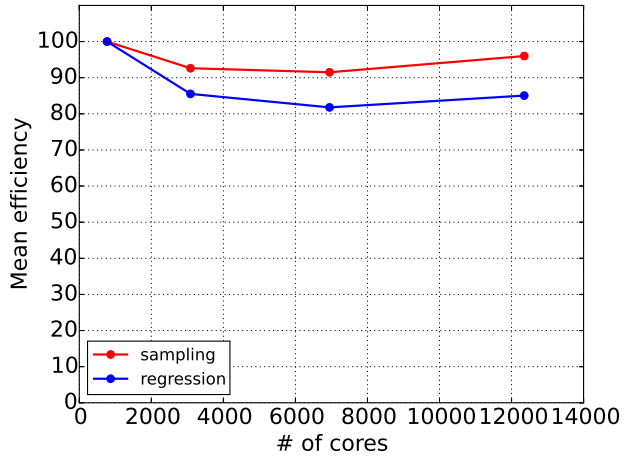

Figure 5. Weak scaling for the server-client implementation.

a single subdomain is completely independent from all the others.

Server-Client Weak Scaling : Weak scaling for the server-client implementation can be setup in two possible ways. The first involves fixing the number of servers, and as the problem size increases, the number of clients is proportionally increased. One drawback of this approach is that it limits the size of the problem that one can tackle, because the number of servers is fixed. This configuration would work well for small problems, but in the limit of the problem size increasing, the memory of the servers would impose a constraint. The alternative is a configuration where the number of clients per server and the amount of data owned by each server is fixed, and the problem size is increased by adding increasingly more clusters. This setting imposes no constraint on the problem size. This is the case that we adopt in this work, as shown in Table 1.

Figure 5 shows weak scaling results for the server-client implementation highlighting the sampling and regression stages. The efficiency for both stays within 80% up to 12k cores, despite the substantial communication inherent in the SC model. Sampling and regression are the stages performed by the clients, while the solution of the boundary maps system and the subdomains updating are done by the servers since they fully own the state.

Comparison between Server-Client and SPMD: In this section, we compare the two implementations, and highlight the performance differences between the two on the same problem. We consider a problem with 48^2 subdomains, overlap of 18 grid cells, and a full domain mesh of 4001^2 grid points. The local grid within each subdomain is about $\sim 100^2$, and the total number of unknowns in the boundary maps system amounts to 742976. For the SPMD case, the number of cores used is 2304. In order to have a fair comparison between the two implementations, we set up the server-client such that the computational power provided by all the clients matches the computational power used in the SPMD case. This implies that the total number of MPI ranks from all clients should equal 2304. The number of servers is fixed to 128, so that each server holds 18 subdomains. Regarding the configuration of the clients, this comparison can be achieved in several ways, but here we consider only one scenario. We choose each client to be made of 1 MPI rank, so that we have a total of 2304 clients, with 48 clients per cluster. This setup closely matches the SPMD case because it maintains a virtual one-to-one relationship between a subdomain and a computational rank.

Table 2 shows the results obtained for the comparison runs between the SPMD and server-client. We only report the results for the two main stages. The first column shows the timings (seconds) collected for the SPMD, while the second column lists those obtained for the server-client, and the last column shows the ratios between them. The overhead due to the communication in the server-client is visible only in the sampling stage while not appearing for the regression. This is because the regression task requires less data transfer than the sampling task. Even though the SPMD implementation is about 30% faster for the current configuration, the better scaling and resilience properties of the SC approach make it better suited for an extreme scale machine with many more cores and significant system fault frequencies, which is the target configuration for this resilient PDE solver approach.

The SPMD implementation serves as a base-line providing a solver environment similar to current applications. From a resilience standpoint, the SPMD model requires the full machine to be resilient. A fully resilient implementation will require additional overhead, and may not address other known weaknesses associated with SPMD models. Some of these include the lack of suitability for emerging manycore systems, the inability to exploit functional on-chip parallelism, and difficulty in tolerating dynamic latencies (20). On the other hand, the SC implementation reduces the overall vulnerability by confining the data to the servers. This

limits expensive hardware and/or data redundancy protocols only to a small part of the machine, allowing the rest to be less reliable, less energy consuming, and, thus, cheaper. This setting aligns well with the vision of future exascale architectures involving hierarchical hardware required to meet energy and cost constraints. For the resilience analysis in the subsequent sections, we will focus on the SC implementation only.

Table 2. Comparison results for the SPMD against the server-client, matching computational power.

Units: seconds	SPMD	SC	SC/SPMD
sampling	127.42	206.77	1.62
regression	1351.57	1375.24	1.02
total runtime	1512.62	2072.72	1.37

6 Resilience Analysis: Description

This section describes the type of faults, and the fault model used for the resilience analysis.

6.1 Soft and hard faults

Various attempts have been made to model and simulate system faults by finding the statistical distribution that best fits the data extracted for real systems, see e.g. (21; 22; 23; 24; 25; 26) and references therein. In general, 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 have an evident impact on the run and the system itself. Silent errors, on the other hand, are more subtle and can go undetected since 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 an important line of future research (5).

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. This is achieved by assuming that if any one of the MPI ranks defining a client dies, the entire client is deemed as dead. This approach is taken to be consistent with the realistic scenario of an entire node failing. A potential alternative scenario is one where a client loses some of its computational ranks, such that its computational power is degraded but it remains active. One real example of this kind would be in the context of multi-core chips with individual cores dying, or a GPU with individual internal processing units failing. We are currently investigating this scenario from an implementation standpoint to explore how degraded clients would affect the runtime. Since standard 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. 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. More specifically, for a given task, this is implemented by performing bit-flips on randomly selected bits of the task's data.

Table 3. 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

6.2 Failure Distribution

In this work, a memoryless Poisson process was chosen as a means to introduce errors. The impact of, e.g., correlated error patterns using more advanced failure models is left for future work. A Poisson process is uniquely defined by a single parameter, namely the rate of failure r, leading to a failure distribution

$$F(t) = \int_0^t r \exp(-r\tau) d\tau = 1 - \exp(-rt). \qquad (6)$$

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 (26). This reference was chosen because it provides an extensive and detailed study on different faults for real systems. Even though the hardware has improved over the years becoming more reliable and less error-prone, in this work we aim at investigating extreme fault conditions in order to see how well the approach performs under such conditions. We extract failure rates by scaling up the results found in (26) 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 3 reports the computed failure rates for each of the fault category target in this work.

From an implementation standpoint, to simulate the occurrence 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. If the total number of tasks involved in the algorithm was known in advance, one potential alternative to simulate faults would be to randomly choose in advance ("offline") which tasks are hit by faults. One advantage of this method is that failures would not be correlated with aspects of the computation, e.g. the longer the runtime, the more likely a fault can occur. However, this "offline" approach is not feasible for our algorithm because

we do not know exactly how many tasks are generated within the algorithm. In our case, e.g. during the sampling stage as described below, the number of tasks is unknown because the sampling is run until a certain number of tasks comes back to the servers. Pre-assigning tasks and failures is thus not feasible. Moreover, from a different viewpoint, we also think it is realistic that the more a computing unit works, the more likely it is that a failure can occur, e.g. due to overheating. We also remark that our implementation does not rely on actively detecting faults. When a hard fault occurs, the client dies (this is simulated by setting it idle) and stops communicating with its server. From a practical standpoint, the server does not have to query if a client is still alive, because the server gathers only the data sent back by clients that are still alive. In this work, we do not support the scenario of a hard fault hitting a client such that it remains alive but works in a degraded fashion. For soft faults, the algorithm does not try to detect them, but simply works with the data that is available.

6.3 Handling Faults

The server-client 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 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 resubmit 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. 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 itself 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 does not contain NaN or Inf. If that is the case, then the task is simply discarded.

6.4 Test Problem

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 setting 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 arbitrarily set to 7 grid cells along both x_1 and x_2. Within each subdomain, the PDE is discretized using a second-order finite difference (FD) approximation. A representative plot of the discretization grid, the resulting decomposition, the subdomains and their overlapping is shown in figure 6.

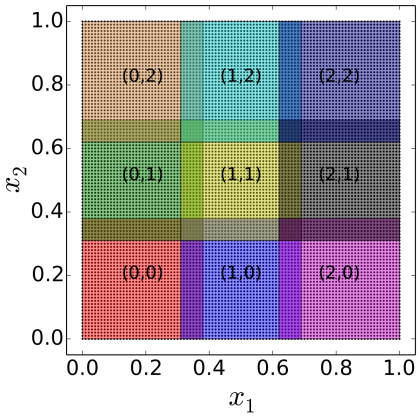

Figure 6. Schematic showing the $n_{x_1}^{(s)} = n_{x_2}^{(s)} = 3$ subdomains partitioning, as well as the underlying $n_x = n_y = 101$ discretization grid.

For each value of the failure rate shown in Table 3, we run an ensemble of $N = 40$ simulations for the $S9g_s37^2$ case. To explore the impact of the problem size, 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. For all the results below, unless stated otherwise, we focus on the case involving one MPI rank being the server, and 62 clients, each consisting of 2 MPI processes, yielding a total of 125 running MPI processes.

7 Resilience Analysis: Results

The first part of the results focuses on the effects of hard, computation and communication faults individually. The second part discuss the results when all three types of faults are present at the same time.

7.1 Hard Faults

The effect of hard faults on the algorithm implementation are first explored using the $S9g_s37^2$ test case. Figure 7 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 corresponding failure rate. 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 shows that all the runs converge regardless of the number of hard faults hitting the "system", thus proving the resiliency of our algorithm with respect to hard faults.

Figure 8 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 successfully completed without being hit by any fault 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

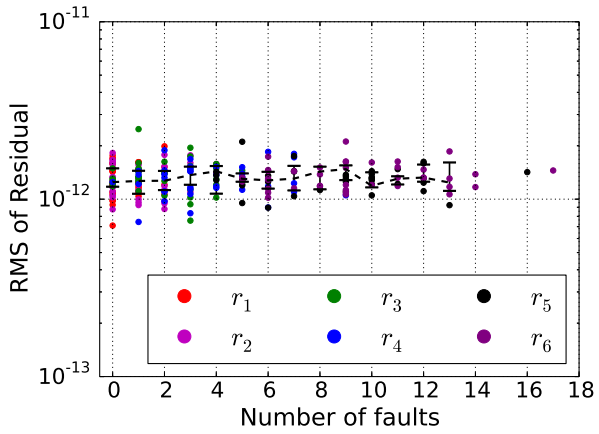

Figure 7. Root-mean-square value of the final residual plotted as a function of the number of *hard* faults. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles. The markers are color-coded based on the corresponding failure rate.

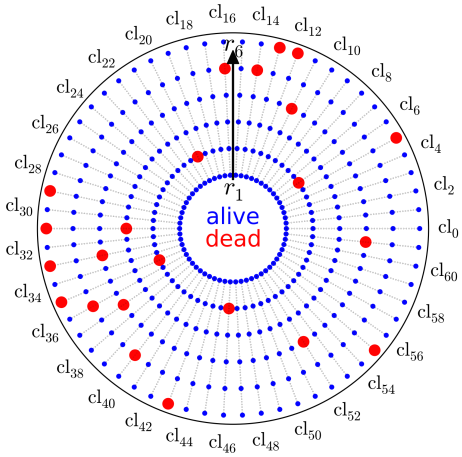

Figure 9. Status of all 62 clients (distributed along the angular direction) for a representative simulation of each of the failure rates r_i, $i = 1, \ldots, 6$. A red marker identifies a client that has died during that run, while a blue marker identifies a client that is alive.

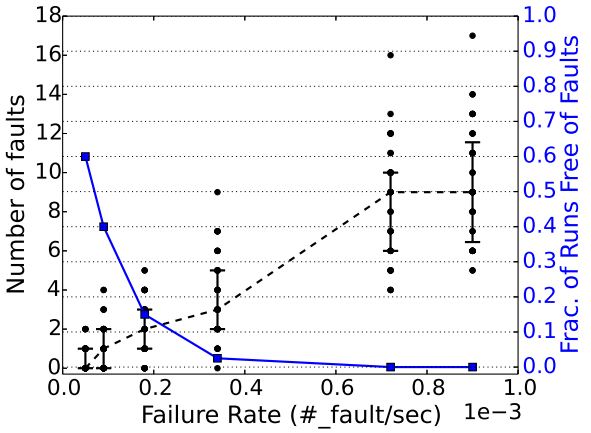

Figure 8. Number of faults for all ensemble runs (black circles) plotted as a function of the failure rate obtained for the *hard* fault case. The errors bars are obtained for the 0.25 and 0.75 quantiles, while the dashed line connects the 0.5 quantiles. The dataset in blue shows the fraction of runs that complete without faults. All results are obtained for the $S9g_s37^2$ case.

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

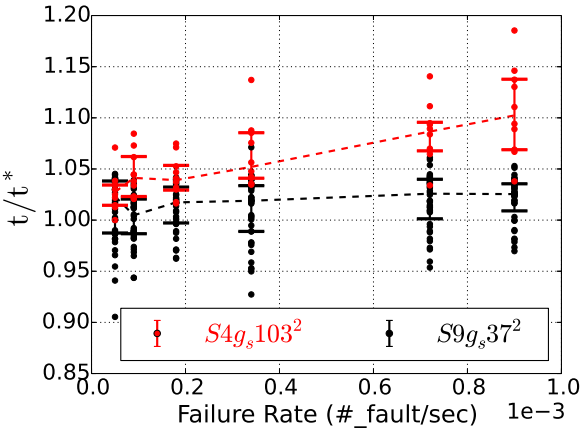

Figure 10. Time per iterations t normalized by t^*, which is the corresponding time for the no faults case, plotted as a function of the number of *hard* faults. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles.

To investigate what is the effect of the faults on the runtime, 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 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 10 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 figure shows that some runs for the $S9g_s37^2$ complete more quickly than the no-fault case. This can be due to the fact that in some cases, the sampling and regression stages happen to have sampled data making individual tasks complete faster, which in turn makes the overall execution faster. 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. Overall, however, the algorithm handles hard faults very well.

7.2 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.

Figure 11 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

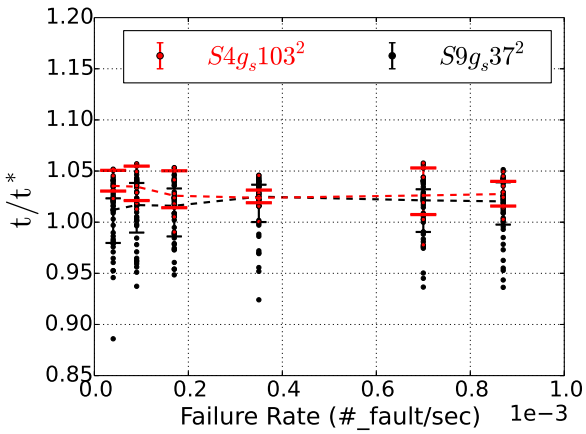

Figure 11. Time per iterations t normalized by t^*, which is the corresponding time for the no faults case, plotted as a function of the number of *computation faults*. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles.

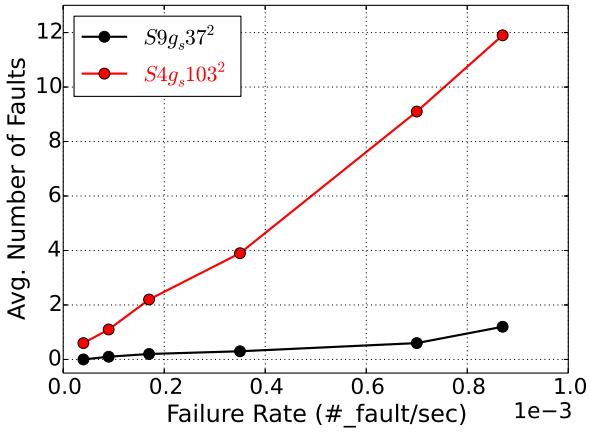

Figure 12. Average number of faults computed over the ensemble runs as a function of the failure rate obtained for the *computation faults*.

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 12 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

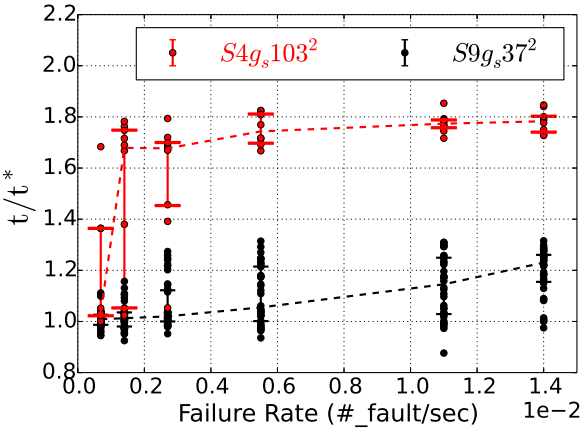

Figure 13. Time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of failure rate for *communication faults*. The errors bars are obtained for the 0.25 and 0.75 quantiles, and the dashed line connects the 0.5 quantiles.

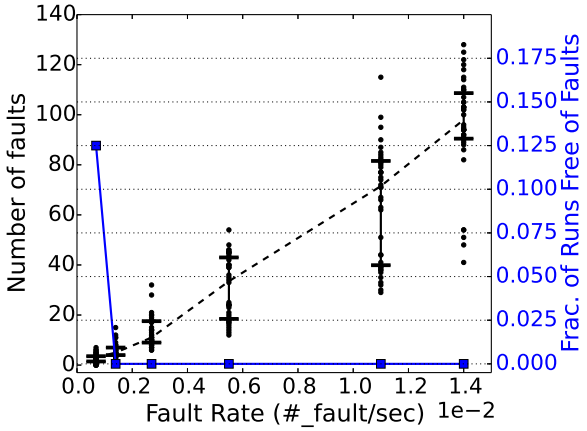

Figure 14. Number of faults for all ensemble runs (black circles) plotted as a function of the failure rate for *communication faults*. The errors bars are obtained for the 0.25 and 0.75 quantiles, while the dashed line connects the 0.5 quantiles. The dataset in blue shows the fraction of runs that complete without faults.

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 11. 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.

7.3 Soft Faults During Communication

We now explore the effects of faults occurring during the communications 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 a task object 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 even the object itself being completely corrupted. This is what we are trying to model with this type of fault. Our approach relies on synthetically corrupting all the data of that object. There is an important distinction to make between messages traveling from the server to a client, and those 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 a task object carrying corrupted data. We remark once again that these are silent errors so the client does not know that the data is corrupted, unless the data is clearly unusable, i.e. it 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 then decides 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, because we might have too many corrupted data, or the magnitude of the perturbation may be so large that the regression solver cannot overcome it. If this is the case, then it is likely that using these data will lead to a bad solution, in which case the algorithm runs 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 13 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. As anticipated before, this is due to the fact that if a task fails, it is rerun multiple times. 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 settle 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 14 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. At such high fault rates, the main conclusion is that the algorithm is very robust to silent communication faults, at the cost of an extra iteration from time to time.

7.4 Mixed Faults

The results discussed above were based on exploring the effect of each type of fault individually. This analysis allowed us to extract some patterns and highlight their individual effects on the runs. We now investigate how the algorithm behaves when all faults are activated 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 15 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%. For the larger case, as was seen in the previous section, an extra iteration is often required for the large fault rates. 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.

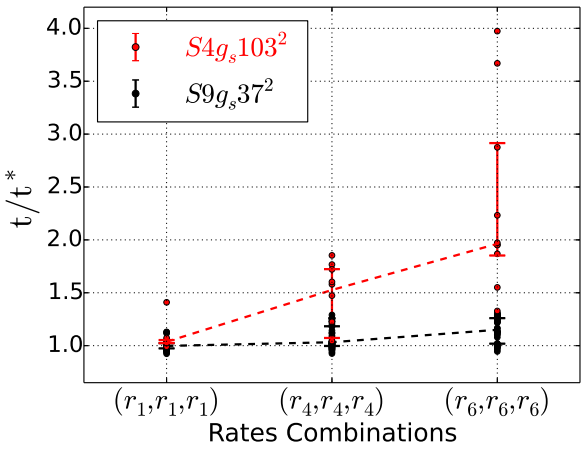

Figure 15. Time per iteration, t, normalized by t^*, which is the corresponding time for the no faults case, as a function of the mixed fault case.

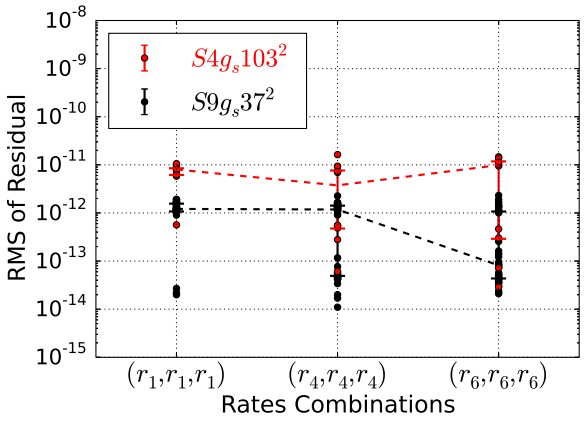

Figure 16. RMS of the residual as a function of the mixed fault case.

We finalize this section showing the resiliency results. To this end, figure 16 shows the RMS of the residual as a function of the mixed fault cases. Even though the residuals show more variability as the fault rates increase, the plot shows that all runs converge with very good accuracy.

8 Conclusions

We presented a PDE preconditioner that is resilient to hard and soft faults, and showed a test involving a 2D 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 discussed two implementations, one based on the SPMD model and one based on a server-client model. The scalability of both implementations was presented, and the main differences were highlighted. The scalability results showed excellent scalability for the major components of the algorithm. The comparison between the SPMD and SC implementations showed that, for the current configuration, the SPMD implementation is about 30% faster than the SC. Despite this result, the better scaling and resilience properties of the SC approach make it better suited for extreme scale applications with many more cores and significant system fault frequencies, which is the target configuration for this resilient PDE solver approach.

The asynchronous server-client 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, mimicking the silent errors that can occur within the network when data is being transmitted. First, we remark that in all cases, the algorithm always converges. 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 communication and computation cost. For the network faults, the effect of the faults is substantial due to the large number of faults happening, as well as the abrupt 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 intuitively expected, this scenario is the most dramatic one, with very large numbers of faults. Regardless, the algorithm converges with good accuracy in all the runs.

Acknowledgements

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.

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.

References

[1] DOE-ASCR. Exascale programming challenges. Technical report, 2011. URL http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ProgrammingChallengesWorkshopReport.pdf.

[2] Ang JA, Barrett RF, Benner RE et al. 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, Piscataway, NJ, USA: IEEE Press. ISBN 978-1-4799-7564-8, pp. 25–32. DOI:10.1109/Co-HPC.2014.4. URL http://dx.doi.org/10.1109/Co-HPC.2014.4.

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

[4] Cappello F, Geist A, Gropp B et al. Toward Exascale Resilience. *International Journal of High Performance Computing Applications* 2009; 23(4): 374–388.

[5] Cappello F, Geist A, Gropp W et al. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations* 2014; 1(1). URL http://superfri.org/superfri/article/view/14.

[6] Bosilca G, Delmas R, Dongarra J et al. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing* 2009; 69(4): 410–416. DOI:{10.1016/j.jpdc.2008.12.002}.

[7] Ding C, Karlsson C, Liu H et al. Matrix multiplication on gpus with on-line fault tolerance. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*. pp. 311–317. DOI:10.1109/ISPA.2011.50.

[8] Du P, Bouteiller A, Bosilca G et al. 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, New York, NY, USA: ACM. ISBN 978-1-4503-1160-1, pp. 225–234. DOI:10.1145/2145816.2145845. URL http://doi.acm.org/10.1145/2145816.2145845.

[9] Chen Z. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*. HPDC '11, New York, NY, USA: ACM. ISBN 978-1-4503-0552-5, pp. 73–84. DOI:10.1145/1996130.1996142. URL http://doi.acm.org/10.1145/1996130.1996142.

[10] Ali M, Southern J, Strazdins P et al. Application level fault recovery: Using fault-tolerant open mpi in a pde solver. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. pp. 1169–1178.

[11] Shye A, Moseley T, Reddi V et al. 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*. pp. 297–306. DOI:10.1109/DSN.2007.98.

[12] Malkowski K, Raghavan P and Kandemir M. Analyzing the soft error resilience of linear solvers on multicore multiprocessors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. pp. 1–12. DOI:10.1109/IPDPS.2010.5470411.

[13] Ferreira K, Riesen R, Ron Oldfield JS et al. Keeping checkpoint/restart viable for exascale systems. Sandia Report SAND2011-6815, Sandia National Labs, 2011.

[14] Hoemmen M and Heroux MA. Fault-tolerant iterative methods via selective reliability. Technical report, Sandia National Labs, 2011.

[15] Rizzi F, Morris K, Sargsyan K et al. Partial differential equations preconditioner resilient to soft and hard faults. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. pp. 552–562.

[16] Sargsyan K, Rizzi F, Mycek P et al. Fault resilient probabilistic preconditioner method for one-dimensional pdes. *SISC, submitted* 2015; .

[17] Li ML, Ramachandran P, Sahoo SK et al. Understanding the propagation of hard errors to software and implications for resilient system design. *SIGOPS Oper Syst Rev* 2008; 42(2): 265–276. DOI:10.1145/1353535.1346315. URL http://doi.acm.org/10.1145/1353535.1346315.

[18] Bridges PG, Ferreira KB, Heroux MA et al. Fault-tolerant linear solvers via selective reliability. *ArXiv e-prints* 2012; 1206.1390.

[19] Engelmann C and Naughton T. Toward a performance/resilience tool for hardware/software co-design of high-performance computing systems. In *Parallel Processing (ICPP), 2013 42nd International Conference on*. pp. 960–969. DOI:10.1109/ICPP.2013.114.

[20] Cascaval C and Montesinos P (eds.). *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25-27, 2013. Revised Selected Papers*, *Lecture Notes in Computer Science*, volume 8664. Springer, 2014. ISBN 978-3-319-09966-8. DOI:10.1007/978-3-319-09967-5. URL http://dx.doi.org/10.1007/978-3-319-09967-5.

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

[22] Lin TT and Siewiorek D. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on* 1990; 39(4): 419–432. DOI:10.1109/24.58720.

[23] Oppenheimer D, Ganapathi A and Patterson DA. 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, Berkeley, CA, USA: USENIX Association, pp. 1–1. URL http://dl.acm.org/citation.cfm?id=1251460.1251461.

[24] Vaidya NH. A case for two-level distributed recovery schemes. In *In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. pp. 64–73.

[25] Sahoo R, Squillante M, Sivasubramaniam A et al. Failure data analysis of a large-scale heterogeneous server environment. In *Dependable Systems and Networks, 2004 International Conference on*. pp. 772–781. DOI:10.1109/DSN.2004.1311948.

[26] Schroeder B and Gibson G. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on* 2010; 7(4): 337–350. DOI:10.1109/TDSC.2009.4.