SAND2017-4099J

Modeling and Simulating Multiple Failure Masking enabled by Local Recovery for Stencil-based Applications at Extreme Scales

Marc Gamell, Keita Teranishi, Jackson Mayo, Hemanth Kolla, Michael A. Heroux, Jacqueline Chen and Manish Parashar

Abstract—Obtaining multi-process hard failure resilience at the application level is a key challenge that must be overcome before the promise of exascale can be fully realized. Previous work has shown that online global recovery can dramatically reduce the overhead of failures when compared to the more traditional approach of terminating the job and restarting it from the last stored checkpoint. If online recovery is performed in a local manner further scalability is enabled, not only due to the intrinsic lower costs of recovering locally, but also due to derived effects when using some application types.
In this paper we model one such effect, namely multiple failure masking, that manifests when running Stencil parallel computations on an environment when failures are recovered locally. First, the delay propagation shape of one or multiple failures recovered locally is modeled to enable several analyses of the probability of different levels of failure masking under certain Stencil application behaviors. Our results indicate that failure masking is an extremely desirable effect at scale which manifestation is more evident and beneficial as the machine size or the failure rate increase.

Index Terms—Parallel processing, stencil computation, resilience, fault tolerance, failure masking, modeling

1 Introduction

Science and engineering applications are increasingly relying on very large-scale simulations to make advances and enable new insights and discovery. This has motivated the need for achieving **exascale** (10^{18} FLOPS) computing capability by the end of the decade [1], [2] or early next decade. The enormous scale and complexity of an exascale system, which would be composed of a very large number of hardware and software components, poses many challenges, one of which is system reliability. On current petascale systems, the observed frequency of the occurrence of process and node failures, described by the mean-time-between-failures (MTBF) metric, is of the order of hours [3]. Anecdotal evidence from production simulations on ORNL's Titan Cray machine showed nine process/node failures in a day, as shown in [4]. This is consistent with the fact that observed hard failures in current HPC centers cluster together in time, resulting in periods of high instability followed by long periods of stability. For example, even though Titan's MTBF is in the order of 7.75 hours, around 30% of the failures occur within an hour of a previous failure while some other failures are separated by 24 hours [3]. While it is difficult to predict what the actual MTBF of an exascale machine will be, some researchers believe that it may be in the order of minutes [1].

- *M. Gamell and M. Parashar are with the Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ 08854, USA. E-mail: {mgamell,parashar}@cac.rutgers.edu*
- *K. Teranishi, H. Kolla, J. Mayo, and J. Chen are with Sandia National Laboratories, Livermore, CA 94551, USA. E-mail: {knteran,jmayo,hnkolla,jhchen}@sandia.gov*
- *M. Heroux is with Sandia National Laboratories, Albuquerque, NM 56310, USA. E-mail: maherou@sandia.gov*

This significant reduction in reliability will directly impact applications, since the typical run time of target scientific applications will be longer than the MTBF. As a result, resilience will be a key design requirement for exascale systems and applications, and fault tolerance techniques will be essential. This has been well documented in reports such as the Blue Waters & TeraGrid 2009 workshop [5], DoE's ExaOSR [6] and the Argonne National Laboratory's 2012 resilience workshop report [7].

Implementing fault tolerance directly at the hardware level has clear advantages, such as allowing the software to assume a reliable substrate. However, there can be a significant cost associated with sustaining such an assumption. First, hardware-based fault-tolerance incurs a one-time cost of developing the techniques and a per-component cost to implement them in silicon. Second, these techniques require other operating costs, for example, to keep the extra circuitry running. An example is the hardware-implemented Error Correcting Code (ECC) technology that is shipped with high-end RAM modules. ECC consumes a certain amount of power to function, which adds to the overall power consumption of the HPC center. Since reducing power consumption is also expected to be a significant challenge at exascale [6], systems may not be able to afford power-hungry hardware dedicated to tolerate failures or may have to trade off performance to improve resilience.

To avoid these associated costs, fault tolerance has to be addressed at other levels. While programming models such as task-DAG-based may naturally tolerate these kinds of failures, in more traditional programming models based on communicating sequential processes (CSP), e.g., MPI, a single process/node failure results in the immediate termination of the entire application. Currently, the recovery from

such failures often involves restarting the job from the last checkpoint, which is stored and retrieved from stable storage. The most widely used strategy for this is a coordinated global checkpoint/restart from stable storage, which incurs data movement and I/O costs and these costs can become overwhelming during periods of machine instability since the time to checkpoint and restart might be longer than the time between failures. Efforts are being made to address this issue by developing runtimes that provide an abstraction of a fault-free system to the application (e.g., MPICH-V [8], redMPI [9]). As suggested by recent studies [10], the system-level techniques behind these frameworks will not be sustainable at extreme scales, necessitating resilience techniques that are application-aware.

In our previous work on the Fenix framework [4], we have demonstrated that it can efficiently enable online global recovery for message passing applications, used in conjunction with application-aware checkpointing, to handle failures occurring as frequently as every 47 seconds. However, every failure triggers a global recovery which requires all surviving processes to first recover the MPI environment collaboratively and then, along with newly spawned processes or spare processes, rollback to the last available checkpoint. While this is advantageous –the failure and the consequent recovery mechanism can be completely transparent to the application– global recovery represents a disproportionate response to an individual failure and has inherent scalability challenges. It would be clearly beneficial if recovery mechanisms can be made more scalable by taking advantage of inherent characteristics of applications.

Numerical solutions of partial differential equations (PDEs) represent a large number of scientific applications. One class of PDE solvers, based on the finite-difference discretization, primarily involves stencil operations and has certain favorable characteristics. The finite difference methodology typically consists of a number of iterations, and within each iteration there are two key phases: (1) computation on local data using the stencil operator to advance state, and (2) communication of ghost points to immediate neighbors. By taking advantage of the stencil-based computations, the nature of the ghost-communications, the resulting propagation of information, local recovery can be implemented in an efficient and inherently scalable way, avoiding global environment repair and enabling localized rollback [11]. In other words, only a constant number of processes/threads, independent on the total number of processes/threads in the job, need to be aware of a process failure.

Such a local recovery approach allows processes/threads that were not affected by a failure to continue with their simulation. Due to the nature of the nearest neighbor ghost communications, only the immediate neighbors of failed processes will be required to take part on the recovery procedure. Processes affected by a failure are replaced by spare processes or re-spawned processes, and their computations are resumed from a previous state. The time required to recover and rollback –the rollback cost corresponds to the work that was computed between the last checkpoint and the moment the failure stroke– will cause immediate neighbors to block during the ghost communication phase, waiting for the recovered processes/threads to catch up [12][1].

A key objective of this paper is to study and model the effect of individual failures and the impact of local recovery as it propagates through the computational system. An important aspect of this modeling is the interaction of multiple failures, separated in space (computational nodes) and time, which may occur during periods of system instability, and in particular the overheads due to their recovery. We demonstrate that, as the effect of one failure and the delay introduced by its recovery is propagating, if a second failure occurs at a distant node such that the delay due to the recovery of the first failure has not reached it, the effective delay due to the recovery of the second failure will be masked by the delay due to first one. Our proposed model shows that, in general, the overhead on the total execution time due to several separated and independent failures can be smaller than the cumulative overhead due to the individual failures – we denote this effect as **failure masking**. We study the probability of the occurrence of failure masking as well as how this probability changes with increasing scales, and the effectiveness of local recovery approaches for extreme-scale systems. The contributions of this paper include: (1) a qualitative study of the applicability of local recovery approaches to Stencil-based parallel applications; (2) a model of the propagation of delays due to local recovery from failures for stencil-based computations for a one-dimensional domain, and its generalization to a three dimensional domain; (3) a quantitative comparison of global and local online recovery for an increasing number of failures during an application execution; and (4) an analysis of the probability of failure masking as a function of the failure density or MTBF as well as the simulation characteristics.

The organization of this paper is as follows: Section 2 details the background and related work on this topic, Section 3 presents the theory behind the local recovery procedure and its constraints, Section 4 introduces a qualitative description of failure masking, Section 5 models the delay propagation and shows that failure masking can be captured with this model, Section 6 contains the results and an evaluation, and Section 7 presents the conclusions.

2 Background and Related Work

There is vast literature on failures in HPC systems, and in particular process and node failures and their characteristics have been well documented, see e.g. [7], [13]. By far the most commonly employed technique for dealing with failures is "Checkpoint and restart" (C/R) [14], [15], [16] in which the application state is periodically saved to stable storage (e.g., using BLCR [17] or application-level checkpointing) in anticipation of a failure. In the event of one failure, a global rollback is performed and the application is restarted with all processes reading from the last globally committed checkpoint. As mentioned before, this process is independent of the number of nodes running the application, or the size of the failures, since even a single process failure will require all processes to perform the rollback. Local recovery

1. The research presented in this paper builds on and extends the concept of failure masking for stencil computations presented as a short paper [12] as well as the preliminary delay propagation model presented in our previous work [11].

alleviates this response by having only the failed processes perform the rollback to the checkpoint.

Checkpoint coordination. When recovering, process-local checkpoints are required to be globally consistent, which is typically ensured using coordination protocols [16]. Such protocols could involve full coordination [18], non-blocking coordination [19], [20] or blocking coordination [21]. While coordination protocols have the advantage of being application-agnostic, being global in nature they incur the overhead inherent in process synchronization. Uncoordinated protocols [22], on the other hand, relax the synchronization requirement thereby reducing overheads and respecting application imbalance. The problem of ensuring global consistency, however, shifts to examining the created checkpoints and, since global consistency cannot be guaranteed on recovery, all processes may end up rolling back all the way to the initial state, i.e. the domino effect. For piecewise deterministic applications [18] uncoordinated checkpoint protocols could avoid the domino effect by leveraging message logging, at the expense of logging all application messages. This requirement is less severe for send-deterministic applications, which only need a subset of all messages to be logged [23], [24], and a technique for recovering only a subset of processes on failure while avoiding the domino effect was presented in [24] based on these ideas.

We propose an approach based on implicit coordination [4], which guarantees global consistency without requiring a coordination protocol by adopting a checkpointing mechanism where selective checkpoints are created at specific points within the application. To enable recovery, a small set of messages, the total size of which is negligible compared to the size of checkpoints for the target applications, are logged.

Our approach differs from traditional uncoordinated checkpointing and message logging in several aspects: (1) in our approach, all created checkpoints are strongly consistent; (2) we use message logging to enable local recovery, while traditional message logging is used to enable global recovery from a set of non-consistent checkpoints; (3) our message logging is local, in-memory, and used only by spare processes substituting the failed processes; and (4) we guarantee that only the failed processes have to rollback. For iterative applications with a stencil communication pattern, and assuming that the uncoordinated checkpoints are consistently created (i.e., the best case scenario), protocols such as [24] are not ideal since any process failure would trigger all other processes to rollback due to the spread of dependencies on orphan and rolled back messages.

Checkpoint storage. Checkpoints are commonly saved to a parallel file system located in centralized resources [16]. This is a bulk I/O operation and can be expensive from a time and energy perspective, the latter becoming even more critical at future extreme scale systems. Alternatives include storing checkpoints in local memory [25], [26], in both local and peer-memory [27], in non-volatile memory [28], in node-local storage (such as SSD) [29], [30], or at different storage layers [31]. Other cost saving options include compression [32], aggregation [33], or both [34]. This paper presents a theoretical framework for local recovery and failure masking, but a sensible local recovery implementation might only require to store checkpoints in the memory of a peer node and still tolerate multi-node failures with high probability, as discussed in [11], [27].

Message logging. Message logging is a subset of event logging. It maintains a log of all messages between processes to enable reexecution of crashed processes at a later time. To guarantee consistency, the application is assumed to be *piecewise deterministic* [18]. Several destinations for the message logs, such as stable storage or the compute nodes' unused main memory, have been studied. Three broad categories are shown in many studies [16]. In *pessimistic* techniques [22], when a process sends a message, the protocol blocks computation and message delivery until the log is written. *Optimistic* techniques [35] try to reduce overhead by logging messages asynchronously. As a result, the domino effect that pessimistic protocols try to avoid is still possible. *Causal* protocols [36] try to take advantage of both optimistic and pessimistic options by ensuring that the log is either at the final destination or at least cached in a neighbor node. For example, a study by Lifflander [37] done on top of Charm++ exploits the commutative property of messages to reduce the overhead of maintaining a message log; this study, however, requires extra constructs to describe the commutativity of message sequences. Message logging has also been used in conjunction with uncoordinated C/R [19], which allows the application to recover to the state immediately preceding failure. MPICH-V [8] implements two pessimistic message logging protocols and a causal version.

Failure overheads towards extreme scale. Since the expected number of failures that a system experiences is proportional to system size, traditional global recovery may not scale to exascale due to prohibitive recovery times. The total overhead to recover the message passing environment due to a failure depends on the size of the system. For example, assume a system with N_S nodes, in which a node has an expected life of $MTBF_N$ seconds. The expected time to failure of the entire system, therefore, will be $MTBF_S = MTBF_N/N_S$ seconds. If we assume that each failure, in average, requires R seconds to recover, in the best case scenario, R is independent of N_S. The total process recovery overhead O_s (in seconds) for an application running T seconds (towards exascale, we can safely assume that $T \gg MTBF_S$) is expected to be the total number of failures ($\lfloor T/MTBF_S \rfloor$) times the average recovery time, R. Hence,

$$O_s = \left\lfloor \frac{T}{MTBF_S} \right\rfloor \cdot R = \left\lfloor \frac{T \cdot N_S}{MTBF_N} \right\rfloor \cdot R$$

If we divide by T, we will obtain the relative overhead:

$$O_{s,r} = \left\lfloor \frac{N_S}{MTBF_N} \right\rfloor \cdot R$$

The above result shows how, in this scenario, the total overhead is proportional to the size of the system. This is suboptimal, since we would like the total failure overhead to be proportional to the actual failure sizes. We show how local recovery enables multiple failure overhead masking. This result enables a high number of failures to appear in

the total time to solution as if a lower number of failures occurred, and we show how this effect becomes more beneficial with higher core counts.

Stencil computations. Datta et al. model and evaluate the intra-node performance of stencil computations using processors with different architectures, with a special focus on modeling stencil compute performance and memory hierarchy impact [38]. Stark et al. study the implementation of a particular stencil mini application using a task decomposition approach [39], which could be extended to include fault tolerance features.

Classifying failure recovery techniques. We classify recovery mechanisms in four broad categories, disjoint two by two. On the one hand, recovery procedures can be **off-line** or **on-line**, depending on whether they require a complete shut-down of the survivor processes or not. Traditional C/R can be considered off-line, while approaches like Fenix [4], LFLR [40] as well as the techniques presented in this paper are considered to implement on-line recovery constructs. In **roll-back** protocols, when failure is detected every process (including the spare processes) reverts to a valid state in the past at which point the computation is restarted from that state. Traditional C/R uses a roll-back recovery mechanism. While Fenix and LFLR use roll-back recovery as well, it can be considered improved, since the latter systems do not need to synchronize the survivor processes before restarting. In **roll-forward** protocols, on the other hand, survivor processes are allowed to reconstruct a valid state without the need to globally revert the state to a previous consistent state. When failure is detected, every process construct a valid state, or potentially valid after some extra computations. Algorithm-based Fault Tolerance techniques [41], [42], [43], as well as the techniques presented in this paper, fall into this category.

3 Local Recovery for Stencil-based Scientific Applications

In this section we elaborate on our local recovery approach and its suitability for stencil computations. To reiterate, by local recovery we imply that (1) only the re-spawned or spare processes have to rollback to the last checkpoint and (2) only the processes that communicate with the failed ones will notice the failure and might be involved in the recovery process. These requirements are in stark contrast with global recovery which requires all processes to be involved in the recovery and to rollback to a consistent checkpoint. As emphasized before, global recovery is costly, inherently non scalable, and often unnecessary. Furthermore, our local recovery process is by definition an online approach, i.e., the running application does not have to be halted or disrupted. In this section, we start by summarizing the relevant computational characteristics of our target applications: stencil computations. We then describe the local recovery approach in the context of this class of applications, its benefits for single and multiple failures, and related challenges.

3.1 Stencil-based Scientific Applications

Our main target applications are iterative solvers with stencil-based domain partitioning and communication properties. The simplest examples of such applications are parallel PDE solvers using explicit finite-difference schemes for both spatial and temporal discretization. It is worth pointing out that even for time-implicit solvers based on different spatial discretizations (e.g., finite-volume or finite-element) there is an inner component of the iterative solver, which requires repeated communication with nearest neighbors, punctuated by a few global reductions, until it converges. In other words, explicit finite-difference PDE solvers are a good proxy for demonstrating the applicability and benefits of local recovery. The parallel implementation typically involves decomposing the domain across processes, and, as described in the introduction, iteratively performs local computation and ghost communication at every time step for each process. Figure 1 illustrates a typical domain decomposition for a block-structured 2-D computational domain. We note that block-structured mesh is chosen for the purpose of illustration and the techniques presented are applicable to unstructured meshes as well. The communication pattern is between blocks on neighboring processes with each process maintaining a "ghost region" corresponding to the width of the stencil at the block edges. The communication step essentially boils down to a "ghost region exchange", as shown for a 5-point stencil in Figure 1.

One example of such an application is the S3D [44] stencil-based combustion application. S3D is an explicit finite-difference based computational fluid dynamics (CFD) code used to perform massively parallel direct numerical simulations (DNS) of turbulent reacting flows using first principles. The code incorporates high order explicit central difference schemes for spatial derivatives and explicit low-storage Runge-Kutta schemes for temporal integration. It employs realistic gas-phase thermodynamic, molecular transport properties and detailed chemical kinetics by interfacing with the CHEMKIN library. While primarily written in Fortran 90 with MPI based parallel domain decomposition, recent versions have incorporated advanced task-based approaches and OpenAcc pragma based kernel acceleration for heterogeneous architectures to achieve good scalability up to 200 thousand cores on leading petascale platforms.

As mentioned above, some stencil computations require to periodically perform global operations that involve all processing elements, such as reductions on the partial results. These operations may be used, for example, to check for errors or analyze partial results. For example, in production runs of the S3D simulation, global reductions occur approximately every 16 minutes [4]. The models described in this paper target the execution ranging from one such collective operation until the next. It is assumed that this time period is long enough for several failures to occur. However, the models and studies presented in this paper still apply, without loss of generality, to cases where this assumption does not hold, for example, where collective operations are performed in an asynchronous manner [45].

3.2 Local Recovery, Challenges and Benefits

This section describes the advantages and challenges of performing local recovery for stencil computations.

Scalable runtime recovery. After a failure, the environment does not need to be recovered globally, and only the

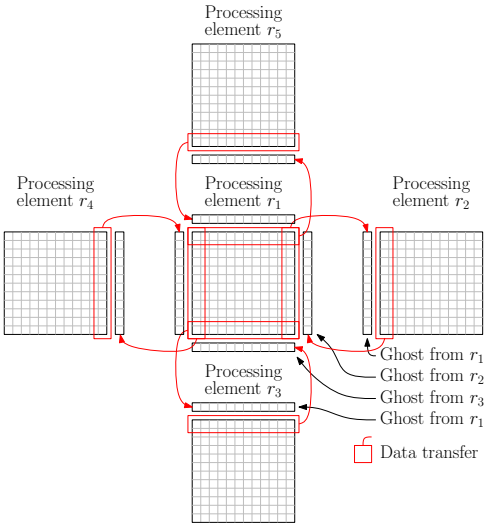


Fig. 1. A portion of a 2D domain is distributed across five processing elements, r_1 through r_5, featuring the communication of cell points between r_1 and its four neighbors [12]. Note how r_1 maintains a copy of the domain distributed in its neighbors in a special buffer, called the ghost buffer.

newly spawned processes have to rollback to the last checkpoint. In other words, only a constant number of processing elements (independent on the total number of processing elements) need to be aware of a multi-process failure. This is a desirable property that allows the implementation of scalable constructs, since the scalability of recovery depends on the failure size, and not on the machine size.

Consistency and determinism. Since the surviving processes need to communicate with the recovered processes and vice versa, and only the recovered processes are forced to roll back, an inconsistent scenario may arise. In order to offer consistency guarantees and deterministic results, the models in this paper assume that messages sent since the last checkpoint are logged in sender memory. Since the size of these messages is a small part of a checkpoint –equivalent to the borders of the domain– the cost of creating and storing the message log are assumed to be negligible. Assuming a square 2D domain of size $O(N)$, the total message log size is $O(\sqrt{N})$, while for a similar cubical 3D domain, the total message size is $O(\sqrt[3]{N}^2)$. For example, for a 3D 7-point stencil computation only six messages need to be stored. Furthermore, iterations creating checkpoints do not require extra space for the message log, since this information is already embedded in the data structures of the application and an optimized recovery can extract the messages (i.e., ghost points) from the checkpoint (i.e., application domain).

Delay propagation. When one or multiple processes fail and are recovered locally, the state of the immediate logical neighbors is more advanced than the state of the recovered process. This is due to the re-execution required at the recovered processes as a result of their rolling back to a previous step. This implies that neighboring processes will not be able to advance past the timestep they were computing when the failure occurred, since they will be required to wait for the ghost regions that failed processes were calculating before failing. The immediate neighbors of the aforementioned neighbors, however, will be able to calculate one extra time step before being required to wait. As a result, the recovery delay does not affect the entire domain immediately, but affects logically closer processes first; the delay diffusively propagates across the domain, resulting in wave-like shapes when represented in diagrams such as Figure 2b (previously presented by our previous work [12], included in this paper for self-containment).

4 Failure Masking

The described delay propagation behavior implies that, for stencil computations using large number of processes that recover from a failure locally, a second failure may occur at processes distant from the first failure that have not been affected by it. In this scenario, the delay of recovering from the second failure will start propagating following a similar wave-like behavior, as shown in Figure 2c. At a certain point, the delay propagation waves will 'merge', effectively masking the smallest of the delays, as if it had never happened. This effect, **failure masking**, contrasts with the behavior of multiple failures using global recovery techniques, in which the recovery effect of failures that are distant in time will always be additive. Figures 2d and 2e show the same effect but with three and seven failures, respectively. It can be observed that the total time to solution of Figures 2b through 2e is equivalent. A failure, however, can also occur at a processing element that has been already delayed by a previous failure, depending on factors such as the communication frequency or the machine size. An example can be seen in Figure 2f, where the end-to-end execution time is equivalent to a case with two failures that are recovered globally. In case failure masking cannot be considered as the delay propagates faster than failure arrivals, local recovery is still more advantageous than global recovery, since it does not require all processes to rollback and compute lost iterations. In that case, while a particular processing element is waiting for the failure to propagate, it can reduce the energy and power consumption of its CPU by scaling its frequency and voltage. The scalability of this effect makes it extremely important for future extreme-scale machines which are expected to have lower MTBF, since the higher the number of processing elements in the system, the longer it will take for the failure recovery delay to spread, effectively increasing the opportunities of multiple failures to occur and mask each other.

The following sections study the failure masking effect in more detail by modeling propagation delays due to local recovery for stencil-based computations, and simulating different failure combinations to determine specific probabilities.

5 Modeling Delay Propagation

Stencil-based computations and how they can benefit from local recovery are described in Section 3, while failure masking is summarized in Section 4. In this section, we present a mathematical model of the execution of stencil-based applications that capture the effects described above. In particular, we present recurrence relations that can effectively model the execution time and delay propagation

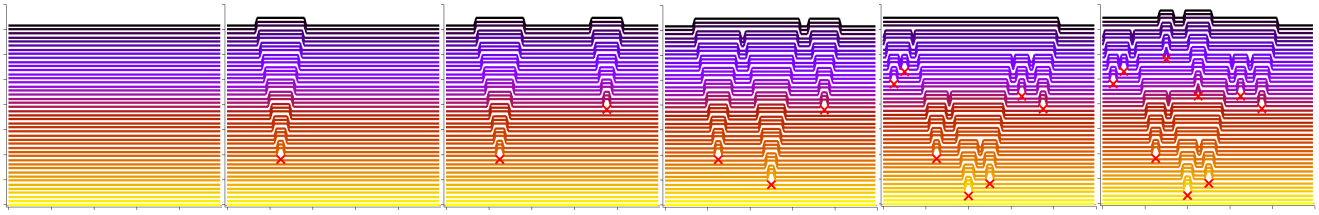


Fig. 2. Delay propagation due to local recovery for a stencil computation on a unidimensional domain [12]. The X axis in each subfigure depicts the processing element number that is computing a particular cell in the domain. The Y axis represents the wall clock time and each horizontal line depicts a particular iteration or timestep, i.e., the moment when a certain processing element finishes computing the timestep. Therefore, a completely horizontal line implies that all processing elements finish computing a timestep at exactly the same time without any imbalance, while failures (marked with red crosses) cause an imbalance (as a result of the recovery delay) at certain processing elements, which is propagated through the domain in subsequent timesteps. Note how all failures in Figures 2b through 2e mask one another, maintaining the total overhead in the end-to-end time to solution constant.

patterns for multi-dimensional stencil-based computations experiencing failures that may mask each other due to local recovery.

Section 3.1 described how processing elements in parallel stencil-based codes typically exchange ghost regions with their logical neighbors. The following recurrence relation can be used to model this behavior for a 7-point three-dimensional stencil:

$$
\begin{aligned}
T(i,p_x,p_y,p_z) =& T_{local} + T_{sync}(i,p_x,p_y,p_z)\\
T(0,p_x,p_y,p_z) =& 0\\
T_{local} =& \begin{cases} T_{it} + T_{Comm} + r & \text{if no failure, or}\\ T_{failure,local} + r & \text{if failure}\end{cases}\\
T_{failure,local} =& T_{rollback} + T_R\\
T_{sync}(i,p_x,p_y,p_z) =& \max\Big(\max_{a\in\{-1,1\}} T(i-1,p_x+a,p_y,p_z),\\
& \max_{a\in\{-1,1\}} T(i-1,p_x,p_y+a,p_z),\\
& \max_{a\in\{-1,0,1\}} T(i-1,p_x,p_y,p_z+a)\Big)
\end{aligned}
$$

In the previous model, we are assuming each processing element is assigned to compute a set of cells in a 3-D domain: p_x, p_y, and p_z are the three components of each processing element's position in the decomposed domain. $T(i,p_x,p_y,p_z)$ represents the wallclock time when the processing element in coordinates (p_x,p_y,p_z) finishes the computation of timestep i. Note that, to advance an iteration, each processing element needs to synchronize with the immediate logical neighbors. The time of this synchronization is represented by the $\max()$ term in T_{sync} since it is assumed that a processing element needs to wait for their logical neighbors to complete their previous iteration before proceeding. T_{local} is the execution time of the local computation, where T_{it} models the time to locally advance the simulation and T_{Comm} models the time to transfer data (excluding synchronization among sender and receiver) in the event no failure occurs. T_R represents the time to recover in the event of a failure at processing element (p_x,p_y,p_z). $T_{rollback}$ is a random variable uniformly distributed between 0 and T_{it} and indicates the rollback overhead of a failure. The term r describes other delays that can occur, such as performance variation among processing elements, and can be used, for example, as the maximum value for a random performance noise generator. Its value is, therefore, expected to be much smaller than T_{it} or T_R. Some applications typically checkpoint regularly every defined number of iterations. This scenario can be approximated by the model by, for example, averaging checkpoint time per iteration and adding the result to T_{it}. Other applications do not need to communicate every iteration. In order to study the behavior of an application that communicates every N_C iterations, the model can be adapted by multiplying T_{it} by N_C. In that case, the simulated timestep would be indicated by iN_C rather than i.

To further extend this model beyond a 7-point 3-D stencil, only T_{sync} needs to be changed to accommodate the new number of neighbors.

We assume that in order to start the computation at a timestep, the communication phase of the prior iteration needs to be completed. This assumption holds for optimizations that are applied in state-of-the-art implementations, such as overlapping the computation of interior points with the exchange of cells in the borders with the logical neighbors. These implementations require to stall any further computations when all local points have been computed and new versions of ghost points need to be fetched. This case is covered by the model since T_{local} contains T_{Comm}, the time to communicate with the neighboring processing elements –communication that will be delayed (in T_{sync}) if a fast processing element needs data that has to be fetched from a delayed neighbor processing element.

The goal of this model is to understand how delay is propagated through the domain due to the synchronization required between logical neighbors. The cost of this synchronization after a failure is in the order of the sum of process recovery time and rollback time. This cost will be much higher than the latency of high-speed interconnects, which allows the model presented to consider the costs of processing element substitution as negligible. For example, in order to recover from a process failure, new resources are typically allocated far away from the failed process. This implies that, in some network topologies, the communication latency between the recovered processing element and their logical neighbors increases. However, this increase when using high performance interconnects is negligible compared to the total cost of process and data recovery, and can be masked by high-quality implementations that

overlap communication and computation.

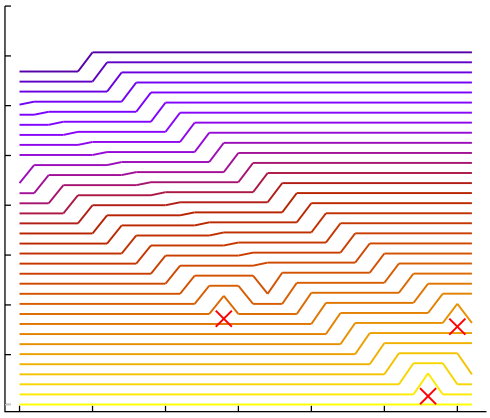
As empirical results presented in our preliminary work [11] corroborate, the results and assumptions in this model accurately represent the behavior of both simple 1-D PDEs as well as complex stencil-based codes such as the S3D application.

6 Evaluation

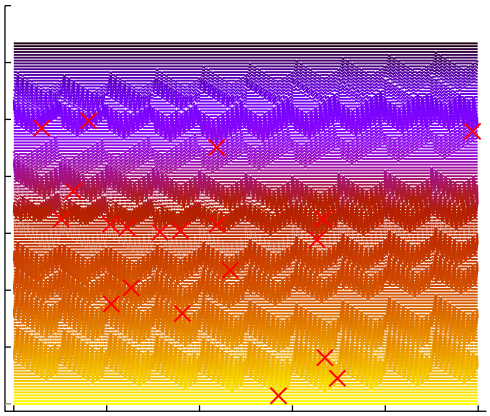
Based on the concepts within the model presented in Section 5 we have implemented a simulator to study the progress of each processing element in the event of failures. The simulator implements a skeleton of a stencil application and, iteratively and sequentially, computes the execution time at each processing element for every simulated timestep. The simulator can randomly generate failures based on the desired failure count, the desired MTBF, or the desired failure rate per processing element. Failures are generated uniformly across the processing elements, and uniformly across the run time by using C++'s `std::mt19937` random number generator and `std::uniform_int_distribution` random number distribution.

In the analyses presented in this section, we abstract time as 'time units' (which could be seconds or minutes) and resources as 'processing elements' (which could represent threads, processes, sockets, or compute nodes). A discussion about how processing elements and time units may be defined can be found later in this section.

Unless otherwise specified, these studies assume stencil-based simulations running on future extreme scale machine with processing elements organized as a cube. Such a cubic organization provides the least advantage for failure masking since any other organization would require more iterations for delays to propagate across the domain, increasing the probability of failure masking. Production runs of S3D running on Titan assigned subdomains consisting of $30 \times 40 \times 30$ cells (i.e., a total of 36000 cells) to each core, and each iteration required around 5 seconds to complete.



(a) 1D simulation, 32 iterations



(b) 3D simulation, 100 iterations

Fig. 3. Execution pattern obtained through a simulation based on the presented model. Figure 3a represents a uni-dimensional stencil and uses 32 processing elements, while Figure 3b represents a three-dimensional stencil and uses one thousand processing elements. The figures have the same meaning as Figure 2: the X axis represents the sequence number of the processing element (e.g. MPI rank), the Y axis represents the wall clock time, and horizontal lines represent the completion of a particular iteration or timestep.

6.1 Propagation of a Multi-Failure Recovery Delay on 1-D and 3-D Simulations

These simulations do reproduce the progress of each timestep as presented in Figure 3. For example, on Figure 3a a particular 1-D 3-point stencil execution is simulated with $r = 0$ and 32 processing elements. One can observe how the first and the third failure propagations are merged. We observed similar results from the experiments injecting failures in a real 1D stencil code (see Figure 7 in [11]). Another example can be seen in Figure 3b, in which a 3-D 7-point stencil on a $100 \times 100 \times 100$ domain are simulated while twenty failures are injected. The failure propagation in the 3-D case is not as trivial to see as in the 1-D case, since the plot's X axis indicates processing element number, and the mapping from processing element number to 3-D simulated position is not visually trivial. However, as in the 1D case, similar results can be observed between the modeled 3-D case and actual executions (see Figure 9 in [11]).

6.2 Local Recovery and Failure Masking on a 3-D simulation

Using the simulator described in the previous subsection, we experiment with the statistical overhead of a 3-D execution on $100 \times 100 \times 100$ processing elements for 100 timesteps and an increasing number of injected failures.

For each failure injection count, we ran the simulation for 10048 times. In every sample, we generate a new timestamp and processing element number for each failure injected. We then plot the minimum, first quartile, median, third quartile, and maximum overhead on the execution time for local recovery and global recovery in Figure 4.

In case of the global recovery model, we apply the same delay factor $T_R + T_{rollback}$ to all the processing elements when a failure occurs in one (or many) of the processing elements. This may be optimistic as it does not consider the potentially higher overhead for MPI communicator recovery required when done in a collective manner, as reported in

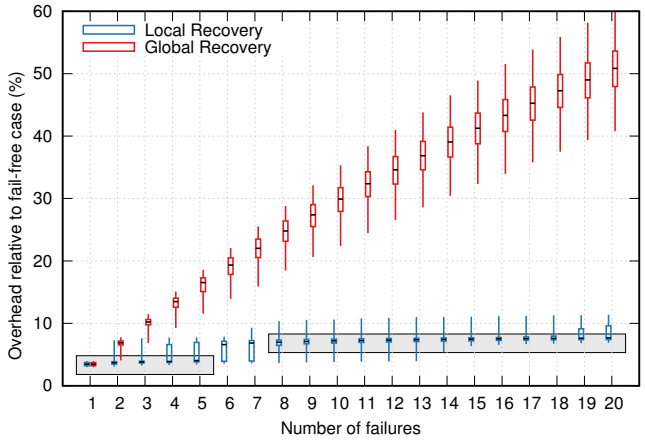


Fig. 4. Recovery overheads for local and global recovery obtained from simulations based on the model for the parallel 3D stencil code running on $100 \times 100 \times 100$ processing elements. In this plot, each candlestick represents the minimum, maximum, median, first and third quartiles overhead of 10048 simulations. Each simulation runs 100 iterations with $T_{it} = 100$ and $T_R = 300$.

our previous work [46]. We do, however, apply the same delay effect to both cases for fair comparison.

The model-based simulation shows the delay from multiple failures is masked by local recovery, reducing the recovery overhead down of the global recovery case by almost an order of magnitude. In this particular case, we can see two differentiated cases: (1) several failures mask each other as if only one single failure occured, and (2) several failures mask each other and appear in the end-to-end run time as if two failures happened. These two cases are indicated in Figure 4 by two horizontal rectangles (shown as gray in Figure). In the first case (1), the median case from injecting 1 failure through 5 failures is extremely similar to the case of globally recovering from a single failure, as indicated by the first gray area. In the second case (2), the median stays around the same value from 8 up to 20 failures recovered locally, as indicated by the second grayed area. Note how in this second case, all medians are comparable to the cost of globally recovering two failures. In the 19- and 20-failure case, however, we can observe how the third quartile increases approaching the 3-failure global recovery mark (10%); this effect can be observed in the 4- and 5-failure case as well. Around the 6- and 7-failure mark, the median overhead for the local recovery case transitions from being comparable to one failure (grayed area on the left) to being comparable to two failures recovered globally, which indicates that the executions with 6 or 7 failures will, in this case, either mask as one failure or as two failures (i.e., with overheads due to recovery the same as those for one or two failures) with high probability.

Conclusion. Figure 4 compares best-case, optimal global recovery (i.e., recovery overhead considered equal to local recovery) with local recovery for the sake of determining impact of more than one failure when using both techniques. It can be observed how, in the case of global recovery, the failure overhead is additive, but, in the case of local recovery, the overhead of multiple failures is much smaller due to the failure masking effect.

6.3 Break-Even Analysis

The result of failure masking is that the effect of multiple failures on the application runtime is the same as that of a single failure. This happens when a failure at a process or node occurs before the delay due to the local recovery of another separate process or node failure has reached it. In this case, the propagating delays from the local recovery of the two failures merge and, since the result is not additive, it appears as if only one failure occurred. Note that this effect would also occur if more than two failures occurred and the resulting delays from their local recovery merged similarly. However, as the number of failures that occur within a specific window of time increases, the probability that the propagating delays due to their local recovery will merge as describe above reduces. For example, the second process or node may fail after it has experience the delay due to the failure of another process or node. In this case, the delays due to the local recovery of the two failures will add.

In this experiment, we explore the break-even point, in terms of the number of failures within a specific time frame, at which failure masking stops occurring. In other words, given a number of processing elements, we determine the probability $p_{be,1}$ that a certain number of failures occurring during a window of time, can mask each other. This study calculates $p_{be,1}$ for different failure counts, thus providing an upper limit on the total number of failures that an application can handle so that the overhead is the same as that of one failure.

We further extend the study beyond the failure masking effect of a single failure, and generalize it to understand the failure density thresholds $p_{be,n}$ for masking failures as if n failures occurred, for $n > 1$. In particular, for practical purposes we will show results for $n = 1, 2, 3, 4$.

Definitions and Assumptions. We want to study the probability that the total overhead of an execution in which a certain number of failures N_F occur within a time frame of T_T will be the same as the overhead when a single failure occurs. We assume a certain failure-free execution time $T_T = N_{it}T_{it} + \lfloor \frac{N_{it}}{N_C} \rfloor (T_{Comm})$, where N_T is the number of processing elements, and that the N_F failures are distributed uniformly in both space (from processing element 0 to processing element $N_T - 1$) and time (from 0 to T_T). We also assume that N_{it} is the number of iterations, and N_C is the number of iterations between communication.

Layered Propagation Delay. It can be inferred from the model in Section 5 that the communication is performed in layers: the immediate neighbors of the failed processing element f_i (which can be considered as the originating layer $l_{i,0}$), can be considered $l_{i,1}$; the immediate neighbors of $l_{i,1}$ that are not in $l_{i,0} \cup l_{i,1}$ (i.e., they have not yet noticed the effects of the failure) can be considered $l_{i,2}$, etc. Therefore, a propagation delay due to the local recover of a failure f_i will only affect processing elements in the layer $l_{i,n}$ after $n \times N_C$ iterations.

The results from this study (i.e., the values for $\{p_{be,1}, p_{be,2}, p_{be,3}, p_{be,4}\}$ for different scenarios) are summarized in Figure 5 and Figure 6, described and analyzed in Section 6.4.

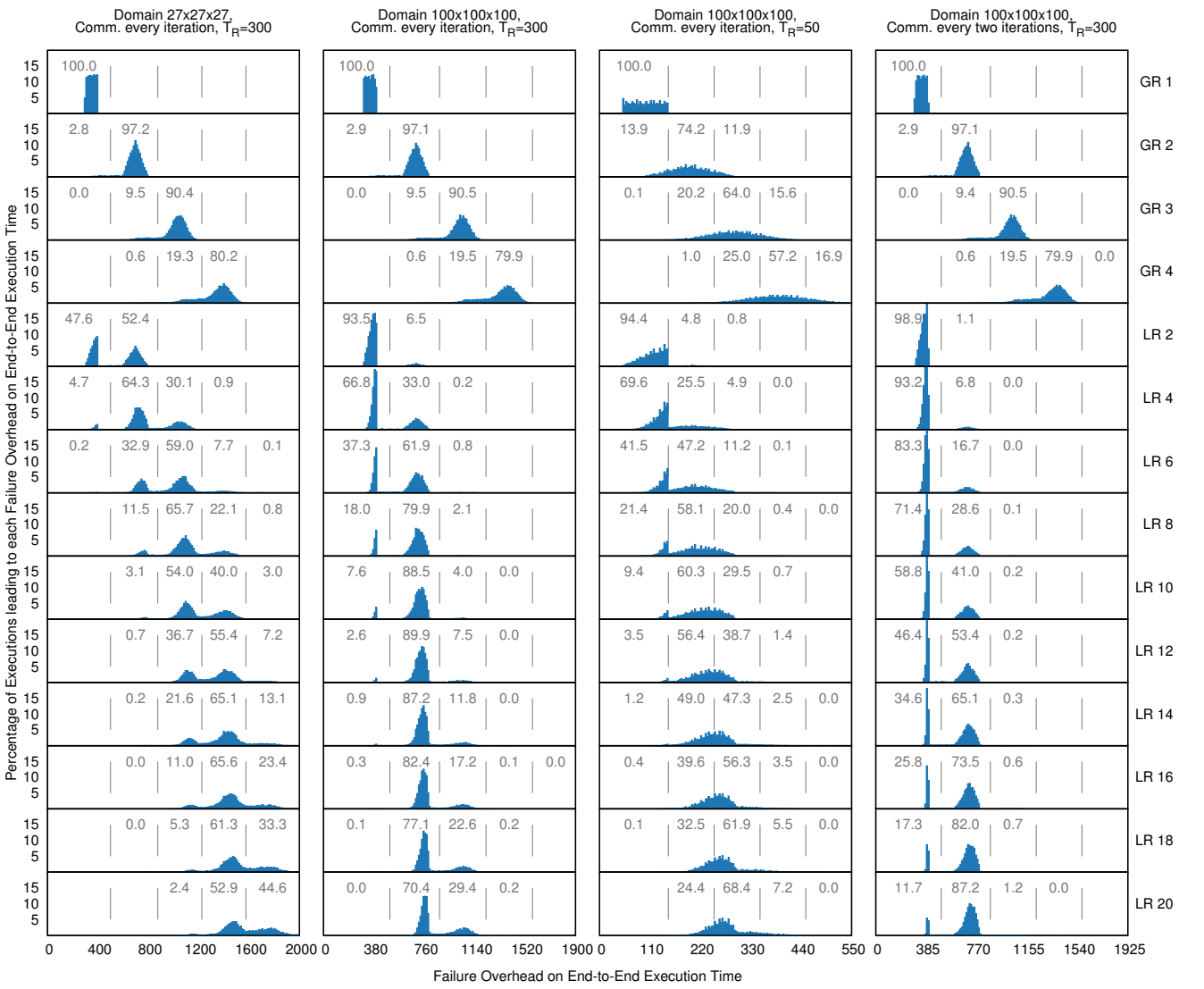


Fig. 5. Histogram of failure overheads of four configurations. For each configuration, fourteen histograms are shown: four with Global Recovery (GR1 to GR4) and ten with Local Recovery (LR2 to LR20). The number i in GRi or LRi indicates the number of failures injected in tests shown in a particular histogram. The histogram for LR1 is identical to the GR1. Each histogram, which contains 10048 samples, represents the overhead of recovering a random number of injected failures. The only variation between samples is the failure position in space and time, each following an independent uniform distribution. The base, failure-free test takes 10,100 time units. Vertical lines attempt to separate the parts of the histograms that have overheads comparable to those with one, two, three, or four failures recovered globally (respectively represented by the four top rows of histograms). The numbers in between those vertical lines indicate the percentage of samples that fall between two horizontal lines which is equivalent to $p_{be,i}$ in LRi. These experiments were done simulating 100 iterations with $T_{it} = 100$ and $T_{Comm} = 1$ (in the cases of communicating every iteration) or $T_{Comm} = 2$ (in the case of communication occurring every two iterations).

6.4 Failure Overhead Distribution for multi-Failure Global and Local Recovery

Figure 5 shows the histogram of failure overheads of four simulations. All simulations perform $N_{it} = 100$ iterations of $T_{it} = 100$ time units each. The basic time to communicate is 100 times smaller than the iteration time: $T_{Comm} = 1$ time unit. The leftmost domain is mapped onto $27 \times 27 \times 27 = 19,683$ processing elements. If we assume a processing element is a full Cray XK7 node (16-core processor + GPU), this simulation would be in the order of a machine similar to Titan at ORNL, with $18,688$ compute nodes. The next test –the second from the left– is the same test scaled up $\sim$51 times to 10^6 processing elements. The application domain is decomposed into $100 \times 100 \times 100$ sub-domains, which are then mapped to the 10^6 processing elements logically organized as a $100 \times 100 \times 100$ cube. According to Top500, Linpack performance on Titan is 17.59 PFLOPS. An Exascale machine has to be around $\sim$58 times more powerful and, hence, this second test can be seen as a hypothetical exascale-level simulation in which nodes are similar to those in Titan[2]. The first two tests were evaluated with a recovery time of $T_R = 300$ time units to simulate a production-level scenario [4]; in the third test the recovery

2. While in reality exascale-type nodes may include many more cores and components, we believe this simulation is valuable due to the fact that the simulated domain will indeed be $\sim$51 times larger than one that fits in Titan.

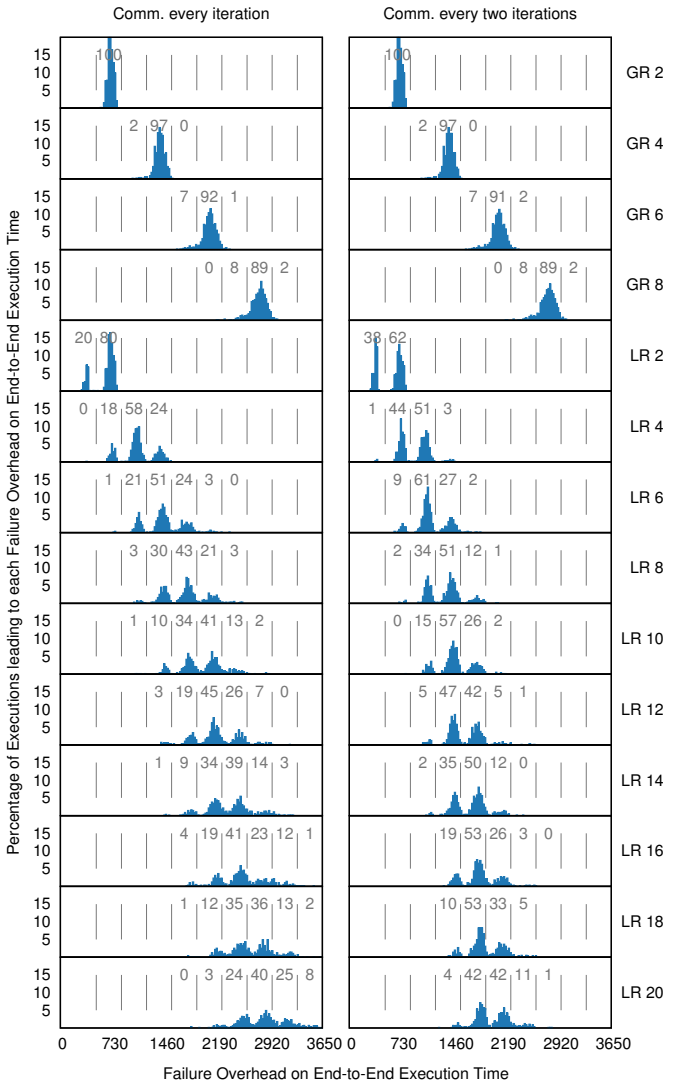


Fig. 6. Histogram of failure overheads of two configurations. In both cases, experiments simulate 1000 iterations of a domain decomposed into $100 \times 100 \times 100$ processing elements and use $T_R = 300$. Otherwise, the configurations and format are identical to those in Figure 5.

time is reduced to $T_R = 50$ time units to simulate the use of optimized recovery. Finally, the rightmost test, again with $T_R = 300$ time units, is a simulation with halved communication frequency (i.e., $N_C = 2$ time units), while the time to communicate is doubled (i.e., $T_{Comm} = 2$ time units).

For each of these configurations, a set of histograms are shown: four for Global Recovery (for one, two, three, and four failures randomly injected), and ten for Local Recovery (for two, four, six, ..., twenty failures injected). Each of these histograms represent the overhead imposed by the respective failure recovery mechanism across 10048 samples[3]. The only variation per sample was the failure position in space and time (i.e., the timestamp of every failure and the processing element it affected) and, as discussed before, the failures were injected following two independent uniform distributions for space and time.

As expected, the histogram of the overhead for globally recovering from a single failure forms a rectangular shape with a minimum at 300 time units (i.e., T_R) and a maximum at 401 time units (i.e., $T_{it} + T_{Comm}$), centered at the average overhead of 350.5 time units. The grey number '100' indicates that 100% of the samples (i.e., all 10048 samples) fall before the first horizontal grey line at position 500. The histogram for locally recovering a single failure in all cases is the same as the one labeled 'GR1' and, hence, not shown.

In the case of globally recovering two failures, 2.8% of the cases fall before the first grey line, i.e., have an overhead similar to the one as 'GR1'. This makes sense, since there is a small probability of both failures occurring during the same iteration (i.e., start iteration $I_i \rightarrow$ failure $\rightarrow$ restart I_i $\rightarrow$ failure $\rightarrow$ restart $I_i \rightarrow$ finish I_i). However, in a large majority of cases, the overhead of globally recovering from two failures will be distributed around an average of 701 time units (double the 'GR1' case). This effect can be seen in Figure 5 in the triangular left-tailed shape of the histogram in all global recovery cases with more than one failure.

The first thing one notices when analyzing the local recovery from two failures on a smaller machine is that 47.6% of the times both failures masked each other as one failure. The remaining 52.4% failures did not mask each other and therefore provide the same overhead as that for global recovery. It is interesting to note, however, that (1) for the four-failure local-recovery case, 69% of the cases masked providing overheads equal to two failures or less and (2) 55+% of the cases when injecting 20 failures masked as four failures or less. That is a 5-fold overhead reduction in highly volatile environments or at times when failure bursts occur.

The situation further improves when considering the larger exascale-level simulation (second case in Figure 5). In almost 70% of the cases, tolerating 4 failures locally will have the same overhead as tolerating a single failure. In this scenario, 70+% of the samples mask as two failures, when injecting 20 failures.

When reducing the recovery time to less than T_{it} (third case in Figure 5), the histograms for global recovery will start to overlap. This poses a challenge for this analysis, since the separation in the local recovery cannot be easily mapped to an equivalent number of failures recovered globally. In this case, we observed that, once again, the histogram for 'GR1' is a square shape of width $T_{it} + T_{Comm} = 101$ time units starting at $T_R = 50$. Therefore, we concluded that in the 'GR2' case, the great majority of samples would be between the values of 151 and 252, which is true ($\sim$75% of samples). The same happens with 'GR3' and 'GR4' and, as a result, we decided to set the division lines at 151, 252, 353, and 454 time units. With these divisions, we can see that the pattern is very similar to the second case with $T_R = 300$, specially up to 'LR6'.

Finally, the right-most subfigure in Figure 5 is the same test as the second subfigure with a halved communication frequency. This can be achieved, for example, by doubling the ghost region size and exchanging the ghost points every two iterations instead of every iteration. If we compare, for example, the injection of ten failures with local recovery in both cases, we can observe that $\sim$60% of the samples

3. These simulations were done using a 64-rank MPI job with 157 samples per job: totaling $64 \times 157 = 10048$ samples

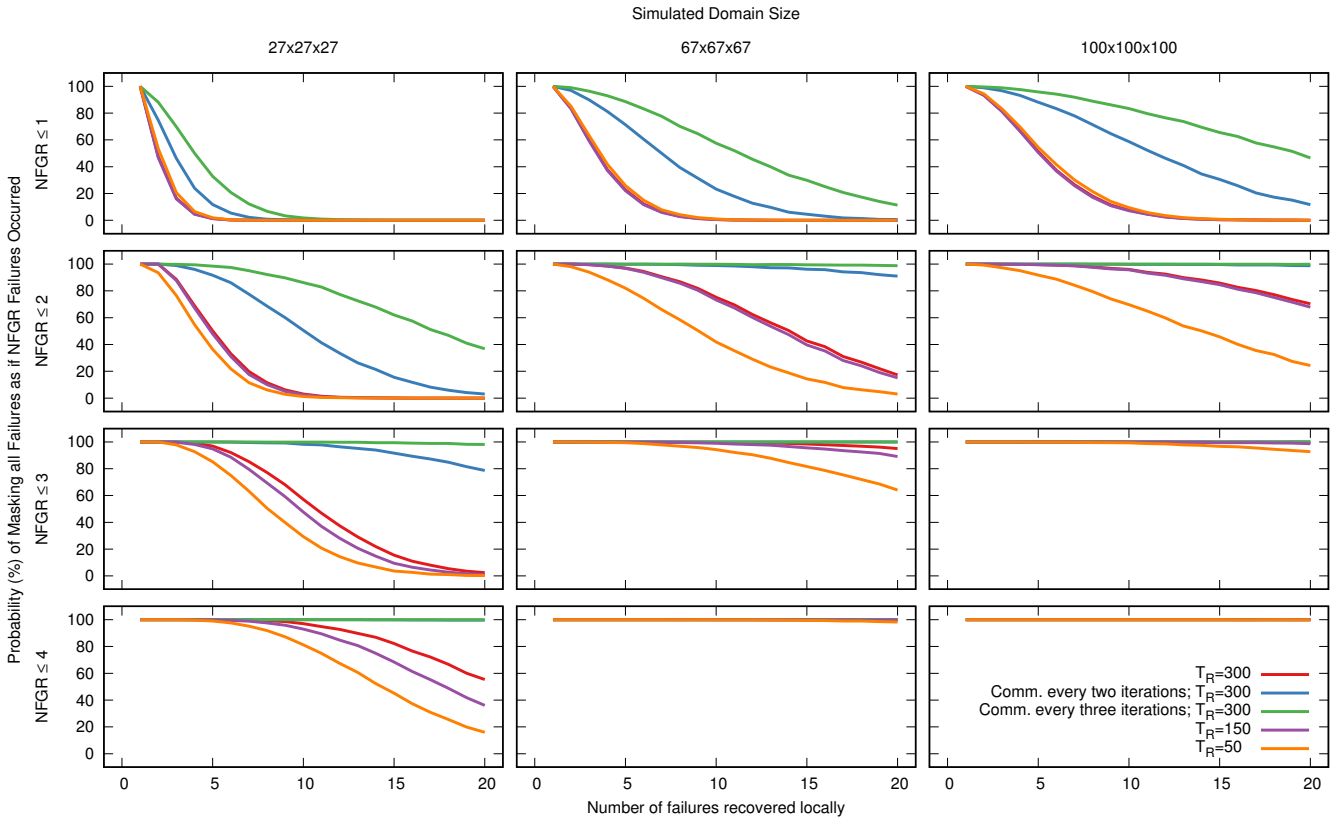


Fig. 7. Probability of masking multiple failures as a single failure (top row of plots), as two or less failures (second row of plots), as three or less failures (third row of plots), or as four or less failures (last row of plots). The Y-axis indicates the probability, from 0 to 100%, that a particular number of injected failures can be masked. The X-axis depicts the total number of failures injected. The leftmost column of plots are simulated with a $27 \times 27 \times 27$ mesh of processing elements, the middle column is simulated with a $67 \times 67 \times 67$ mesh, and the right column is simulated with a $100 \times 100 \times 100$ mesh. The five experiments simulated 100 iterations with $T_{it} = 100$ and $T_{Comm} = 1$ (by default, communicating every iteration). In cases where communication frequency is halved or divided by three, T_{Comm} is increased to 2 and 3, respectively, to account for the extra communication cost. Each trend line connects a total of 20 points with each point evaluated at unit intervals from 1 through 20 along the X-axis.

mask failures as a single failure with half the communication frequency, as opposed to <8% in the standard test. We can also observe that when recovering locally from 20 failures, 98.9% of the samples mask as two failures or less with a halved communication frequency, while this percentage drops to 70.4% in the standard test.

Figure 6 explores two extra configurations by injecting failures at a much smaller density. In particular, we inject the same number of failures as in Figure 5 and increase the total execution time 10 times by simulating 1000 iterations. Focusing on the left subfigure we observe, for example, how six failures are masked as four or less in 73% of cases, or how twenty failures are masked as eight or less with 67% probability. The right subfigure, which plots the same experiment while decreasing communication frequency, shows an increased advantage: six failures are masked as four or less in the great majority of cases, while twenty failures are masked as six or less with 88% probability.

Conclusion. Figure 5 and Figure 6 decompose the simulation samples in a full histogram for a more in-depth analysis than the error-bar representation shown in Figure 4. It also goes one step further by studying the impact of different application behaviors (e.g., domain and machine size, iteration length and count, communication duration and frequency, or different recovery times) on the probability and level of failure masking. Figure 5 and Figure 6 reaffirm the conclusions drawn in Section 6.2, regardless of different application behaviors: (1) the overhead of multiple failures is additive in the case of global recovery, while (2) failure masking enabled by local recovery reduces such overhead by several orders of magnitude.

6.5 Failure Masking Probability

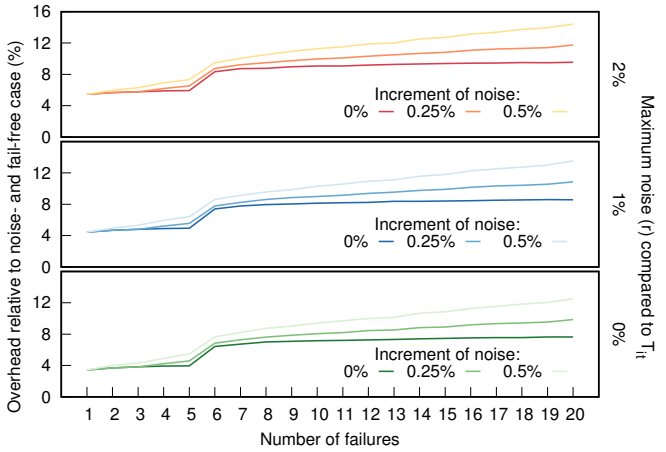
Figure 7 plots the cumulative probability of a certain number of failures (X axis) are masked as 1, 2 or less, 3 or less, and 4 or less failures for five different simulation configurations and three different system sizes. In order to compute the probabilities for a specific configuration and number of failures (from 1 through 20), we repeat each simulation 10048 times varying failure locations in both space and time. We then calculate the percentage of these repetitions in which the number of failures injected are masked as $\{1, 2, 3, \text{or } 4\}$ failures. The left column of plots can be interpreted as a current extreme-scale machine, while the right-most plot can be considered an exascale-level machine. In this plot, the higher a line is, the most beneficial for failure masking a particular experiment will be. We can observe that the recovery time is not extremely impactful in the ≤ 1 and ≤ 2

cases for the $27 \times 27 \times 27$ machine but, as expected, it does negatively impact other cases: shorter recovery times (which are definitely desirable in order to minimize the overhead in end-to-end time) decrease the probability of failure masking of high failure counts.

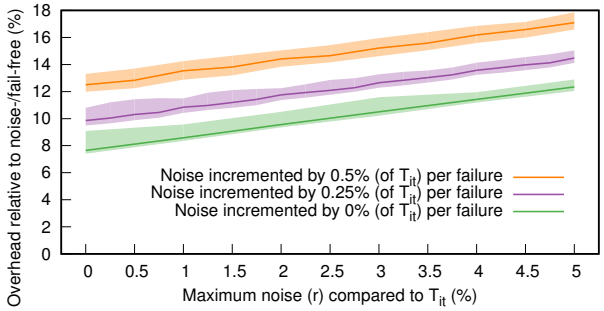
Another observation that can be extracted from the $27 \times 27 \times 27$ machine experiment is that, regardless of the experiment configuration, ten failures will mask as four or less in at least 80% of the cases. If we focus now on the experiments in which communication is done every two or three iterations, we can observe that, with local recovery, 50% or more of the cases will mask ten failures as two or less, and 80% or more of the cases will mask twenty failures as three or less. This demonstrates the enormous benefit of reducing the communication frequency in failure masking.

In the case of 10^6 processing elements, 60% or more of the cases with communication frequency reduction mask ten failures as a single failure, while without communication frequency reduction, in ∼50% of the cases five failures are masked as a single failure. Except in the experiment with recovery overhead of $T_R = 50$ time units, 70% of the cases mask twenty failures recovered locally as two or one. Finally, it is important to note that 95% or more of the cases will mask twenty failures recovered locally as three or less failures.

The experiments have been performed assuming a uniform distribution of failures across space and time. While they take into account temporal correlations (e.g., failure bursts), they do not consider spatial locality. This effect, described by Gupta et al. [47], can be explained by the fact that some failures are triggered by hardware components shared by multiple compute nodes, e.g., network components, power supplies, cooling subsystems. Gupta et al. show, for example, that when a node failure occurs in Titan, the probability of any one of ten subsequent failures (called the correlation window) ocurring in the same physical cage is 9.42% rather than the 1.67% to be expected if failures were not correlated. Similarly, the probability of a node failure in the same node's location is 5.71% (rather than the expected 0.05%). In our study we must exclude same-node failures, as we never reuse a node after it has failed. We leave as future work the study and quantification of the impact on failure masking probability of spatial failure correlations. Certain decrease in the probability can be expected, but since the correlation windows are long and the absolute probability of failures occurring in other parts of the machine compared to the probability of them occurring in the locale is relatively high, the authors do not expect different conclusions, especially in larger machines. If the study does indeed show a significant decrease in failure masking probabilty, an interesting optimization may be to map logically close parts of the domain to compute nodes that do not share hardware components. This optimization would trade off network locality for an increase on masking probability. In contrast, to avoid the spatial failure correlation effect, Gupta et al. propose to quarantine the resources neighboring a failed node for a window of time right after a failure. From a stencil application's perspective, this technique implies that each node failure is promoted to a blade/cage/cabinet failure. How this strategy impacts masking probability is another interesting direction for future work.



(a) The three scenarios show how three base values for r (0%, 1%, and 2%) impact the overhead with an increasing number of failures. In each case, "Increment of noise" is the percentage (relative to T_{it}) that is permanently added to the parameter r after a failure has been observed and recovered.



(b) Three scenarios are shown in which performance variation increases as failures are recovered (a total of 20 failures are injected). Shaded areas show observations within the first and third quartiles.

Fig. 8. Effect of performance variation (i.e., noise) on the total overhead compared to a failure-free and noise-free execution. Execution parameters are set as in Figure 4 and solid lines represent the median of 10048 repetitions.

6.6 Impact of Performance Variation

As explained in Section 5, performance variation often impacts the total execution time, and has been captured in our model by the parameter r. Due to failure recovery, processes or nodes may be re-located within the machine topology, which may impact the total execution time. To capture this effect we increase the performance variation after each failure has been recovered.

Figure 8a shows how an increasing number of failures is masked for different values of r and while increasing r after each failure. The sudden increase when moving from five to six failures is the same effect explained when describing Figure 4. Results show that noise does not affect the probability of failures being masked (notice that in the three subfigures the 0% line is almost flat). Results also show that, even with larger amounts of per-failure noise increments, the recovery overhead of multiple failures can still be masked. When compared with noise-free results of global recovery (see Figure 4) we can see that failure overheads are significantly reduced due to masking.

Figure 8b studies the overhead caused by an increasing

amount of performance variation on the total, end-to-end time when injecting and recovering from 20 failures locally. Figure 8b shows what is the total overhead while increasing the r parameter as failures have been recovered. In particular, after each failure, the r parameter is increased, in each of the three cases, by an amount equivalent to $\{0, 0.25, 0.5\}\%$ of T_{it}.

6.7 Defining Time Units and Processing Elements

The previous simulations have been set up in terms of generic time units and generic processing elements, without actually defining these. We will now try to map these two logical, generic concepts to their real counterparts.

Current estimates for exascale MTBFs vary significantly, so this paper tries to make as few assumptions as possible about what they might be, and therefore we provide a generic time unit that can be adjusted accordingly to represent reality. In what follows, we discuss two possible definitions for time units (1 second and 0.1 seconds) and their implications in terms of the failure rate. Note that a recent study on the failures of current systems highlighted the locality of failures: machines experience long periods of times without receiving any failures (up to three times the MTBF) while a great percentage of failures occur much more frequently than the MTBF (for example, 30% of failures on the Titan Cray XK7 at ORNL occur within one hour of a previous failure, while its MTBF is ~ 7.75 hours) [3]. Therefore, in this paper we consider the burstiness of failures more important than the average MTBF, and focus on making HPC centers still usable during periods of high unreliability.

First, consider that a time unit equals a second, which implies that an application iteration would last 1.6 minutes. While this might seem long, our experiences indicate that it is possible for some complex simulations. As a result, a standard, failure-free simulation will take 10,100 seconds, or about 2.8 hours. During this period of time, we injected a total number of failures ranging from one to twenty, equivalent to bursts of failures ranging, on average, from 168.3 minutes to 8.4 minutes.

Second, if we scale down a time unit to be 1/10th of a second (i.e., 0.1 seconds), each iteration would take 10 seconds to complete (which is in par with our anecdotal experiences) and the total execution time would be 1010 seconds ($\sim$17 minutes). The failures, in these cases, would simulate bursts of failures coming, on average, from every 16.8 minutes to every 50.5 seconds. The application parameters for this case are consistent with executions of the S3D combustion simulation [44] on current systems [11].

Furthermore, we have not defined how a processing element maps to a physical component. One of the options is to consider each processing element to be an entire compute node that require the network to communicate with other processing elements. In the case of the Titan Cray system at Oak Ridge, two compute nodes share the same Gemini ASIC and, therefore, a first option could be to consider a processing element as a group of 16 cores and a GPU. In this case, each of the processing elements may be composed of several processes or threads that would be communicating as well. A second option could be to consider each core in a compute node as a different processing element. This would imply that the difference in the communication cost depending on the location of the cores (i.e., cores in the same NUMA domain will communicate much faster than cores in different compute nodes) is hidden or averaged within T_{it} or T_{Comm}, or that it is considered negligible compared to the cost of recovering from a failure and, hence, does not affect the impacts of failure masking. In this scenario, the $67 \times 67 \times 67 = 300,763$ mesh of processing elements case can be considered similar to the total number of cores on Titan (i.e., 299,008 cores).

7 Conclusion

This paper presented a model for the delay propagation due to local failure recovery in stencil-based computations. The paper also modeled failure masking that can occur during local recovery in stencil-based computations, where the delays due to the local recovery of multiple, independent failures propagate slowly throughout the application domain and can merge in a non-additive way, so that the total overhead experienced by the application is the same as a much smaller number of failures. The paper presented an evaluation based on the model, and studied the probability of this phenomenon happening and the factors affecting it. The studies show that failure masking becomes more effective as the system and application scales increase.

The presented model accurately and consistently reproduces results from previous experimental studies [11]. Furthermore, in this paper we have compared the worst-case overheads due to local recovery with average case overheads due to global recovery for an increasing number of failures, and have shown that local recovery can be significantly beneficial with reasonable confidence levels. Finally, we have also provided an analysis of the probability of several levels of failure masking for different application profiles, using experiments injecting 10,000+ different failure combinations.

The techniques evaluated in this paper help stencil computations to more effectively run on extreme scale systems with increased levels of unreliability, as anticipated at exascale. This research also revisits the co-design discussion regarding the trade off between performance/cost and resilience. Assuming a hypothetical future machine with tunable resilience levels (for example, using power caps), the research presented in this paper indicates that stencil-based computations can obtain higher overall performance or lower cost by disabling resilience mechanisms related to process/node failures and leveraging local recovery and failure masking.

Our ongoing and future work includes (1) exploring how the conclusions in this paper apply to other classes of applications (beyond stencil computations) and (2) modeling and understanding optimal sizes for ghost regions so that delays due to local recovery propagate slower.

Acknowledgments

The authors would like to thank Josep Gamell and Robert Clay for interesting discussions related to this work. The research presented in this work is supported in part by National Science Foundation (NSF) via grants numbers ACI 1339036, ACI 1310283, CNS 1305375, and DMS

1228203, by the Office of Advanced Scientific Computing Research, Office of Science, of the US Department of Energy through the SciDAC Institute for Scalable Data Management, Analysis and Visualization (SDAV) under award number DE-SC0007455, RSVP award via subcontract number 4000126989 from UT Battelle, the ASCR and FES Partnership for Edge Physics Simulations (EPSI) under award number DE-FG02-06ER54857, and the ExaCT Combustion Co-Design Center via subcontract number 4000110839 from UT Battelle. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI^2). 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.

REFERENCES

[1] J. Dongarra and et al, "The International Exascale Software Project Roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011. [Online]. Available: http://hpc.sagepub.com/content/25/1/3.abstract

[2] S. Amarasinghe and et al, "ExaScale Software Study: Software Challenges in Extreme Scale Systems," DARPA IPTO, Air Force Reserach Lab, Tech. Rep., Sep. 2009.

[3] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 25–36.

[4] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014.

[5] D. S. Katz and et al., "Fault Tolerance for Extreme-Scale Computing Workshop, Albuquerque, NM - March 19-20, 2009," Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-312, Dec. 2009.

[6] P. Beckman, R. Brightwell, B. R. de Supinski, M. Gokhale, S. Hofmeyr, S. Krishnamoorthy, M. Lang, B. Maccabe, J. Shalf, and M. Snir, "Exascale Operating Systems and Runtime Software Report," US Department of Energy, Technical Report, Dec. 2012.

[7] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, and et al., *Addressing Failures in Exascale Computing*. U.S. DoE, 2013. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1078029

[8] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI," *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 319–333, 2006. [Online]. Available: http://hpc.sagepub.com/content/20/3/319.abstract

[9] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC 2012, 2012, pp. 78:1–78:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389102

[10] M. A. Heroux, "Toward Resilient Algorithms and Applications," in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, ser. FTXS 2013. New York, NY, USA: ACM, 2013, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/2465813.2465814

[11] M. Gamell, K. Teranishi, M. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar, "Local recovery and failure masking for stencil-based applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 70:1–70:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807672

[12] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar, "Exploring failure recovery for stencil-based applications at extreme scales," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 279–282. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749260

[13] M. Turmon, R. Granat, D. Katz, and J. Lou, "Tests and tolerances for high-performance software-implemented fault detection," *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 579–591, 2003.

[14] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[15] J. Hursey, T. I. Mattox, and A. Lumsdaine, "Interconnect agnostic checkpoint/restart in Open MPI," in *Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*, ser. HPDC 2009. New York, NY, USA: ACM, 2009, pp. 49–58.

[16] J. Hursey, "Coordinated checkpoint/restart process fault tolerance for MPI applications on HPC systems," Ph.D. dissertation, Indiana University, Indianapolis, IN, USA, 2010, aAI3423687.

[17] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 494.

[18] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep 2002. [Online]. Available: http://doi.acm.org/10.1145/568522.568525

[19] B. Bouteiller, P. Lemarinier, K. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2003, pp. 242–250.

[20] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[21] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI," in *ACM/IEEE Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2006, pp. 18–18.

[22] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra, "Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery," in *IEEE International Conference on Cluster Computing and Workshops, CLUSTER 2009*, 2009, pp. 1–9.

[23] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello, "SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 2013. New York, NY, USA: ACM, 2013, pp. 8:1–8:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503271

[24] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011, pp. 989–1000.

[25] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. Panda, "A 1 PB/s file system to checkpoint three million MPI tasks," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, ser. HPDC 2013. New York, NY, USA: ACM, 2013, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462908

[26] G. Zheng, L. Shi, and L. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *IEEE International Conference on Cluster Computing*, 2004, pp. 93–103.

[27] G. Zheng, X. Ni, and L. V. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2012, pp. 1–6.

[28] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing Checkpoints Using NVM as Virtual Memory," in *IEEE 27th International Symposium on Parallel Distributed Processing*, May 2013, pp. 29–40.

[29] X. Ouyang, S. Marcarelli, and D. K. Panda, "Enhancing Checkpoint Performance with Staging IO and SSD," in *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, ser. SNAPI 2010. Washington, DC, USA: IEEE Computer Society, 2010, pp. 13–20. [Online]. Available: http://dx.doi.org/10.1109/SNAPI.2010.10

[30] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 2011, 2011.

[31] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 2010. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.18

[32] D. Ibtesham, D. Arnold, P. Bridges, K. Ferreira, and R. Brightwell, "On the Viability of Compression for Reducing the Overheads of Checkpoint/Restart-Based Fault Tolerance," in *41st International Conference on Parallel Processing (ICPP)*, 2012, pp. 148–157.

[33] X. Ouyang, R. Rajachandrasekar, X. Besseron, H. Wang, J. Huang, and D. K. Panda, "CRFS: A lightweight user-level filesystem for generic checkpoint/restart," in *International Conference on Parallel Processing (ICPP)*. IEEE, 2011, pp. 375–384.

[34] T. Islam, K. Mohror, S. Bagchi, A. Moody, B. De Supinski, and R. Eigenmann, "MCREngine: A scalable checkpointing system using data-aware aggregation and compression," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 2012, Nov 2012, pp. 1–11.

[35] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using asynchronous message logging and checkpointing," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, ser. PODC 1988. New York, NY, USA: ACM, 1988, pp. 171–181. [Online]. Available: http://doi.acm.org/10.1145/62546.62575

[36] E. Elnozahy and W. Zwaenepoel, "Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526–531, 1992.

[37] E. M. Jonathan Lifflander, H. Menon, P. Miller, S. Krishnamoorthy, and L. Kale, "Scalable replay with partial-order dependencies for message-logging fault tolerance," in *Proceedings of IEEE Cluster 2014*, Madrid, Spain, September 2014.

[38] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1137/070693199

[39] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, "Early experiences co-scheduling work and communication tasks for hybrid mpi+x applications," in *Proceedings of the 2014 Workshop on Exascale MPI*, ser. ExaMPI '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 9–19. [Online]. Available: http://dx.doi.org/10.1109/ExaMPI.2014.6

[40] K. Teranishi and M. A. Heroux, "Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 51:51–51:56. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642774

[41] K.-H. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[42] F. T. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques," *J. Parallel Distrib. Comput.*, vol. 5, no. 2, pp. 172–184, Apr. 1988. [Online]. Available: http://dx.doi.org/10.1016/0743-7315(88)90027-5

[43] 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*, ser. PPoPP 2012. New York, NY, USA: ACM, 2012, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145845

[44] J. H. Chen and et al., "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science and Discovery*, vol. 2, no. 1, p. 015001, Jan. 2009.

[45] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, "A Case for Standard Non-Blocking Collective Operations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI 2007*, vol. 4757. Springer, Oct. 2007, pp. 125–134.

[46] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra, "Practical scalable consensus for pseudo-synchronous distributed systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 31:1–31:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807665

[47] S. Gupta, D. Tiwari, C. Jantzi, J. Rogers, and D. Maxwell, "Understanding and exploiting spatial properties of system failures on extreme-scale hpc systems," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 37–44. [Online]. Available: http://dx.doi.org/10.1109/DSN.2015.52

Marc Gamell is currently working toward the PhD degree at the Rutgers Discovery Informatics Institute, Rutgers University. He received BS and MS degrees at the Open University of Catalonia, Spain (2010 and 2012, respectively).

Keita Teranishi is Principal Member of Technical Staff at Sandia National Laboratories. He received BS and MS degrees at the University of Tennessee, Knoxville (1998 and 2000, respectively), and a PhD degree at Penn State University (2004).

Jackson Mayo is Distinguished Member of Technical Staff at Sandia National Laboratories. He received a BS degree at the Muhlenberg College (2000) and a PhD degree at Princeton University (2005).

Hemanth Kolla is Principal Member of Technical Staff at Sandia National Laboratories. He received a BTech degree at the Indian Institute of Technology, Madras (2003), an MEng degree at the Indian Institute of Science (2005), and a PhD degree at University of Cambridge (2009).

Michael A. Heroux is Senior Scientist in the Center for Computing Research at Sandia National Laboratories and Scientist in Residence at St. John's University. He received a BA degree at Saint John's (1984), and MS and PhD degrees from the Colorado State University (1986 and 1989, respectively).

Jacqueline Chen is Distinguished Member of Technical Staff at the Combustion Research Facility at Sandia National Laboratories and the founding Director of the ExaCT Center. She received a BS degree at the Ohio State University (1981), an MS degree at the University of California (1982), Berkeley, and a PhD degree at Stanford University (1989).

Manish Parashar is Distinguished Professor of Computer Science at Rutgers University and the founding Director of the Rutgers Discovery Informatics Institute. He received a BE degree at Bombay University, India (1988), and MS and PhD degrees from Syracuse University (1994).