SAND2015-0310C

Failure Masking and Local Recovery for Stencil-based Applications at Extreme Scales

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

[†]Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ, USA
{*mgamell, parashar*}*@cac.rutgers.edu*

[‡]Sandia National Laboratories, Livermore, CA, Albuquerque, NM, USA,
{*knteran, maherou, jmayo, hnkolla, jhchen*}*@sandia.gov*

ABSTRACT

Application resilience is a key challenge that must be addressed in order to realize the exascale vision. Previous work has shown that online recovery, even when done in a global manner (i.e., involving all processes), 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. In this paper we suggest going one step further, and explore how local recovery can be used for certain classes of applications to reduce the overheads due to failures. Specifically we study the feasibility of local recovery for stencil-based parallel applications, and develop programming support and scalable runtime mechanisms to enable online and transparent local recovery on current leadership class systems. We show how multiple independent failures can be masked to effectively reduce the impact on the total time to solution. We have implemented these techniques using the Fenix framework, providing mechanisms for transparently capturing failures, re-spawning new processes, fixing failed connections, restoring application state, returning execution control back to the application, and handling duplicate message requests. We deployed Fenix on the Titan Cray-XK7 production system at ORNL and experimentally demonstrate the benefits for local recovery for stencil applications. We then integrate these mechanisms with the S3D combustion simulation, and demonstrate Fenix's ability to tolerate high failure rates (e.g., node/blade/cabinet failures every 5 seconds) with low overhead while sustaining performance, at scales up to 262144 cores.

1. INTRODUCTION

The increasing demands of science and engineering applications continue to push the limits of current extreme-scale systems. As a result, the HPC community is working towards achieving **exascale** (10^{18} FLOPS) by the end the decade [1, 13]. One of the key anticipated challenges at exascale will be the reliability of the system, primarily due to the very large number of cores and components [4, 26, 33]. The mean time between failures (MTBF) for current petascale systems is measured in days (e.g., production runs on ORNL's Titan Cray showed 9 node failures/day, as shown in Figure 1), but it is estimated that the MTBF for an exascale system would be measured in minutes [13]. An important class of failures that must be addressed is process and node failures, including correlation effects. These failures are often recovered by terminating the job and restarting it from the last checkpoint found in stable storage. While coordinated, stable-storage-based global checkpoint/restart (C/R) is currently the most widely accepted technique for addressing processor failures, it is unclear whether this approach will scale to exascale since the time to checkpoint will often be longer than the expected MTBF, and researchers continue to actively address this issue. For example, runtimes that aim to offer an abstraction of a fault-free system to the application have been developed (e.g. MPICH-V [7], redMPI [15]). However, as suggested by recent studies [19], the abstraction of a failure-free machine will not be sustainable at extreme scales, and application-aware resilience techniques will likely be required at exascale.

While programing models such as task-DAG can include resilience features, SPMD and message passing are not designed to handle process failures by default. In previous studies, [16] we have shown how online (i.e., without disrupting the job) global recovery (i.e., involving all the cores in the system in the recovery process) can be used in conjunction with application-guided checkpointing to support high failure rates (i.e., every 47 seconds) for message passing applications using the Fenix framework. In that approach, every failure triggered a global recovery, which required all survivor processes to recover the MPI environment. Then, all surviving processes, along with the newly spawned ones, had to rollback to the last commonly available checkpoint. The advantage of global recovery is that it can be done in a semi-transparent way: the application does not necessarily have to be aware of the failure. However, due to the intrinsic global nature of the recovery algorithms, global recovery present scalability challenges.

In this paper we present the design, prototype implementation, and evaluation of **local recovery** approaches for certain classes of applications in Fenix. Specifically we study the feasibility of local recover for stencil-based parallel applications, which represent a significant set of physical simulations, and develop programming support and scalable runtime mechanisms, to enable online and transparent local recovery on current leadership class systems. In additions to its inherent scalability, local recovery provides several benefits. For example, the environment does not need to be recovered globally after a failure, and only the newly spawned processes have to rollback to the last checkpoint.

The key idea underlying the local recovery approach is as

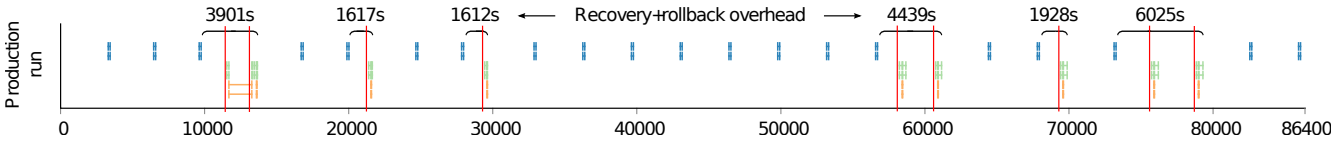


Figure 1: S3D production run on Titan Cray XK7 using 125k cores (December 2013). Note that 9 process/node failures occurred over 24 hours. Using current technique, all failures are promoted to job failures. The fault tolerant technique used was checkpointing (5.2 MB/core) stored in the PFS. Total overhead due to fault tolerance was 31.40% of the total execution time. Blue lines represent checkpoints, light green represent process recovery overhead, orange represent checkpoint load time, and vertical red lines indicate the moment where the node failure occurred.

follows. Stencil-based parallel applications, such as simulations that solve partial differential equations (PDEs) using finite-difference methods, typically consist of a number of iterations (timesteps) with each iteration consisting of two key steps, computation on local data to advance the simulation, and communication with the immediate neighbors. This communication pattern implies that, upon failure, by allowing the rest of the domain to continue the simulation, only the immediate neighbors will be immediately affected by that failure. When a failure occurs, we can substitute the failed process with a spare one, rollback to the last saved state for the failed process (i.e., the last checkpoint), and resume computation for that process. In this paper we show how the effect of the failure will slowly propagate through the machine. If subsequent failures occurs at a distant node *before* the original failure delay has spread to that node, we demonstrate that the delay of the second failure will be masked with the delay of the first one. In general, we show that the overhead of several separate failures on the total execution time can appear to be as the overhead of a single failure (**failure masking**).

We have implemented these solutions using the Fenix framework and deployed them on the Titan Cray-XK7 production system (world's second fastest machine as of November 2014) at ORNL. We decided to implement Fenix as a standalone runtime which only dependencies are a C++ compiler and the Cray uGNI library. This has been a non-trivial task. The motivation behind this decision is that fault tolerant versions of MPI, such as ULFM [5, 6], do not fit in our requirements. Specifically, they are not capable to deliver local recovery constructs. We present an experimental evaluation of the effectiveness and scalability of local recovery in Fenix using the S3D [11] stencil-based combustion application. S3D is a highly parallel method-of-lines solver for PDEs and is used to perform first-principles-based direct numerical simulations of turbulent combustion. It employs high order explicit finite difference numerical schemes for spatial and temporal derivatives, and realistic physics for thermodynamics, molecular transport, and chemical kinetics. S3D demonstrates good scalability up to nearly 200K cores, and has been highlighted by [2] as one of five promising applications on the path to exascale.

Our results demonstrate Fenix's ability to tolerate high-frequency dynamically injected node failures while maintaining sustained performance of S3D on scales up to 262144 cores. Our evaluation also explores extreme execution scenarios that may exist at exascale, where node failures occur with high frequency (i.e., as often as every 5 seconds). For example, when injecting node failures every 30 seconds, performance is sustained with 13.75% overhead when compared with a failure-free and checkpoint-free execution. Finally, we show that the programming overhead of using Fenix is low, requiring less than 35 new, changed, or rearranged lines of code in S3D.

Key contributions for this work include: (1) A study of the applicability of local recovery approaches to stencil-based parallel applications, including a model to understand and estimate the propagation of recovery delay due to failures; (2) design and implementation of this approach within the Fenix runtime, and its deployment on the Titan Cray XK7 production system at ORNL; and, (3) an experimental evaluation of local recovery algorithms in Fenix on Titan using the S3D combustion application demonstrating its ability to support sustained performance and scalability in spite of high frequency real node failures.

2. BACKGROUND AND RELATED WORK

Process and node failures and their characteristics have been well documented in [33, 35]. Checkpoint and restart (C/R) [20–22] is the most widely used technique for implementing resilience for HPC systems. In this model, the application state is periodically saved (e.g., using BLCR [18]) so that, upon failure, global rollback can be used to restart the application from the last globally committed checkpoint. This process is independent of the number of nodes affected by the failure, i.e., if a node or process failure occurs, all processes are typically forced to rollback to the previous strongly consistent checkpoint. In contrast, using local recovery in Fenix, only failed processes need to rollback to the previous checkpoint.

Checkpoint coordination. Global consistency of locally-created checkpoints [20] is typically ensured using coordination protocols. Examples of **coordination protocols** include full coordination [14], non-blocking coordination [9,10] or blocking coordination [12]. The main advantage of coordinated protocols is that they are application-agnostic and create globally consistent checkpoints. The major drawback, however, is the overhead due to process synchronization. **Uncoordinated protocols** [8] do not required synchronization during checkpoint creation, thus reducing overheads and allowing application imbalance. However, during recovery, a consistent global state has to be found by examining the checkpoints. As these protocols cannot guarantee checkpoint global consistency on recovery, all processes may end up rolling back to the beginning of the execution, i.e., the domino effect. Uncoordinated checkpointing protocols can leverage message logging to avoid the domino effect for piecewise deterministic applications [14], at the expense of logging all the application messages. For send-deterministic applications, only a subset of all the messages need to be logged [17, 31]. Building on these ideas, the re-

search presented in [17] proposes a technique for recovering only a subset of processes on failure, while avoiding the domino effect. Furthermore, by letting the application indicate when checkpoints are created, **implicitly coordinated protocols** presented in [16] enable the advantages of coordinated protocols (i.e., guaranteed checkpoint consistency) to be combined with those of uncoordinated protocols (i.e., local checkpointing without synchronization).

Our approach in Fenix uses such implicit coordination, and, in order to enable local recovery, it logs a small set of messages. The total size of these logged messages, for the target applications, is negligible when comparing with the size of the checkpoints. However, 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 the failed processes; and (4) we guarantee that only the failed processes have to rollback. Note that using protocols such as those presented in [17] with our target applications (i.e., iterative applications with a stencil-based communication patterns), and assuming that the uncoordinated checkpoints are consistently created (i.e., the best case scenario), a process failure would require all processes of the system to rollback to the last checkpoint because orphan and rolled back message dependencies would extend across all of the mesh. This is not the case with our protocol.

Checkpoint storage. Typically, checkpoints are saved to a centralized parallel file system [20] but may also be stored in local memory [30], in both local and peer-memory [36], in non-volatile memory [25], in node-local storage (such as SSD) [3, 28], or at different storage layers [27]. They may be compressed [23], aggregated [29], or both [24]. In order to enable local recovery, Fenix only needs to store checkpoints at a peer node. Other strategies, such as storing checkpoints in the parallel file system or compressing them would add performance overheads that makes them prohibitive for Fenix, despite their advantages.

Combining optimized checkpointing with global recovery. Systems such as Fenix [16], LFLR [34] and FMI [32] show how advanced in-memory diskless checkpointing can be used in conjunction with global recovery to enable execution in a failure-prone scenario.

3. LOCAL RECOVERY FOR STENCIL-BASED SCIENTIFIC APPLICATIONS

This section presents the local recovery approach and our underlying reasoning for exploring this approach for stencil-based applications. Recovering from failures in a local manner implies that (1) only the re-spawned 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 contrast with global recovery, in which all the processes are involved in the recovery and rollback to the last consistent checkpoint. Global recovery can be costly and presents scalability challenges, and, in many situations, may be unnecessary. Note also that local recovery is by definition an online recovery approach, i.e. the job does not have to be disrupted.

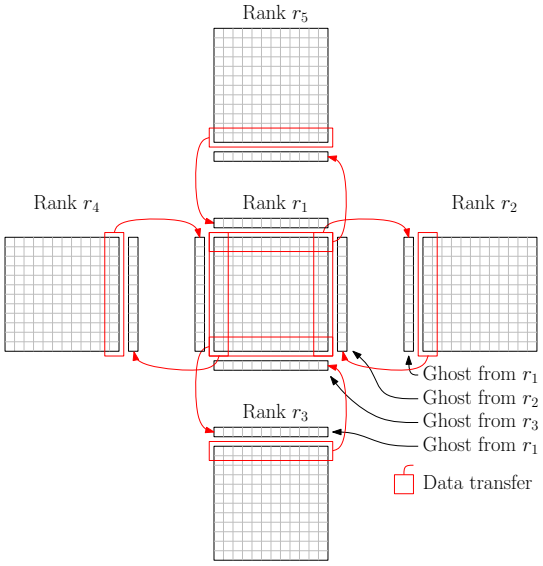


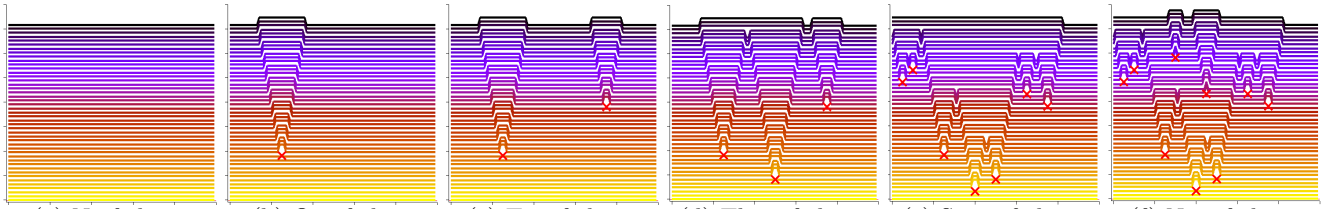
Figure 3: Partitioning of a 2D domain across five processes. This figure shows the ghost region buffer exchange between neighboring processes in a typical implementation of a stencil-based parallel application.

In this section we first describe the key relevant characteristics of the targeted stencil-based applications. We then explore the local recovery approach for this class of applications, its benefits in case of single and multiple failures, as well as associated challenges.

3.1 Stencil-based Scientific Applications

In this work, we target iterative applications with stencil-based domain partitioning and communication properties, such as for example, typical parallel implementations for PDE solvers using finite-difference methods. In these applications, the application domain is typically partitioned using a block decomposition across the processes, and each process perform two key tasks at every timestep: (1) computation on its local data to advance the simulation, and (2) communication with its immediate neighbors that based on the specific stencil used. A typical block decomposition for a 2-D stencil-based application is illustrated in Figure 3. The figure also illustrates the communication pattern between blocks on neighboring processes. In a typical implementation, each process maintains a "ghost region" corresponding to the width of the stencil used around its blocks, and populates this region from its neighbors in a "ghost region exchange" communication step. The exchange shown in Figure 3 is for a 5-point stencil.

Not all scientific applications offer the descrived iterative behavior from the beginning of the execution until the end. Sometimes, collective operations are performed every certain number of timesteps, for example, for analysis, error checking, etc. In this paper we focus the execution between two consequent such synchronizations, and assume that this interval is long enough so that, at extreme scales, several failures can occur within it. For example, in case of the S3D application, this interval is typically every 16 minutes. Our focus is to enable the application to continue the execu-



(a) No failures (b) One failure (c) Two failures (d) Three failures (e) Seven failures (f) Nine failures

Figure 2: Behavior of local recovery for a stencil-based 1-D partial differential equation (PDE) solvers. X axis represents process number (or rank) and Y axis indicates wallclock time. Each line in a figure represents a timestep, and the color of the line represents how advanced the simulation is (i.e., it advances from yellow to dark purple). Each red 'X' represents a failure. A straight line means all processes compute the timestep at the same physical time. When a failure occurs, the recovery delay does not get propagated immediately to the entire domain. In stead, the immediately adjacent neighbor processes are the first to be delayed, which in turn propagate the delay to their immediate neighbors, resulting in the delay eventually spanning across the entire domain. Note how Figures 2(b), 2(c), 2(d) and 2(e) have the same recovery overhead, i.e., as if only one failure occurred, even though they have different numbers of failures. In case of Figure 2(f), however, the total recovery time is equal to sequentially recovering from two failures.

tion between these such synchronizations despite the number of failures occurring and the system size. If this assumption is unrealistic, we can assume, instead, without loss of generality, that the collective operations can be done in an asynchronous manner. Asynchronous collectives are promising, because they can naturally support imbalance between the processes without imposing barrier-like constraints. We leave understanding how local recovery can be beneficial even with periodic blocking collective operations as future work.

3.2 Local Recovery, challenges and benefits

Realizing local recovery for target stencil-based parallel applications, implemented using message passing (MPI), presents several challenges and benefits.

Consistency. As neighbor processes must communicate, guaranteeing consistency in a message passing environment can be challenging. In the approach implemented and evaluated in this paper, we log messages that have been transferred since the last checkpoint, and store them in local memory. Specifically, for the 1D case, only two messages are stored every timestep: the message sent to the node in the right, and the message sent to the node in the left. In the 3D case, 6 messages need to be logged at each timestep. However, note that the overhead of logging the messages is negligible compared to the cost of checkpoint because (1) the checkpoint is several orders of magnitude larger, and (2) message logs are kept in the sender's local memory and no network transfer is required. By storing the messages at the sender side, upon recovery, the re-spawned processes will be able to request the messages again.

Delay propagation. Assume that a node failure occurs while the processes mapped to the node are between iterations C_{i-1} and C_i. Once the failure is detected, the last checkpoint can be fetched from the checkpoint store used to restart the execution of the failed node on either a node from a spare pool or a re-spawned node. While this is happening, the rest of the processes can continue working as usual. The fact that the failed process advanced beyond C_{i-1} guarantees that all their immediate neighbors were also already past this point. Note that, in order for a process to advance beyond a certain communication point, it has to exchange information with their immediate neighbors. This is also true even when the ghost exchange is non-blocking, because sender-based message logging guarantees the availability of the data even when the failure occurs between the data transfer. The iterative and stencil-like nature of the targeted applications will eventually require immediate neighbor processes (i.e., L_1 neighbors) that communicate directly with the failed node to wait. Even though these processes can continue executing the next iteration, it is likely that when they reach the next communication phase (i.e., C_i), the restarted neighbors will not have reached that point yet. Therefore, the immediate neighbors will have to wait. In turn, second-level (L_2) neighbors (i.e., the immediate neighbors of L_1) will be able to continue its execution up to iteration C_{i+1}, and will then be blocked. This is possible because the L_1 processes are waiting at C_i, which means they are not able to exchange data with the L_2 processes at iteration C_{i+1}. In general, kth-layer neighbors would be able to continue until iteration C_{i+k-1} without blocking. This wave-like delay propagation behavior can be seen in Figure 2(b) for a 1-D stencil. While we use 1-D to illustrate the process, this behavior also applies to higher dimensions.

Failure masking. When using a large number of processes, it is possible that another failure occurs on distant processes where the delay from the first failure has not yet reached. In this case, the recovery delay of the second failure will begin propagating from the second location, as see in Figure 2(c). At some point in space and time, the delay of both failures will *merge*. At this point, the total delay will be the maximum of both delays. We call this effect *failure masking*, and an example can be seen in Figure 2(d). This situation is beneficial at large scales, because the impact of several failures on end-to-end execution time will be comparable to that of a single failure. Note that this effect can also happen with multiple failures, as seen in Figures 2(d) and 2(e). Comparing these four figures, we see that the total overhead is the same. Note that the larger the machine is, the more plausible this effect becomes, which is an ideal property for good scalability.

There may be cases, however, where failures occur after the delay of previous failures have already reached the failed node. An example can be seen in Figure 2(f), in which the total execution time is comparable to that of recoverying

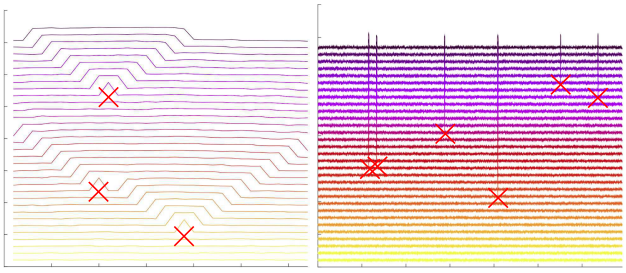


Figure 4: Results for delay propagation obtained from simulations based on the model for 32 cores (left) and 13824 cores (right). Each horizontal line indicates the same timestep across the nodes. X indicates the point of failure.

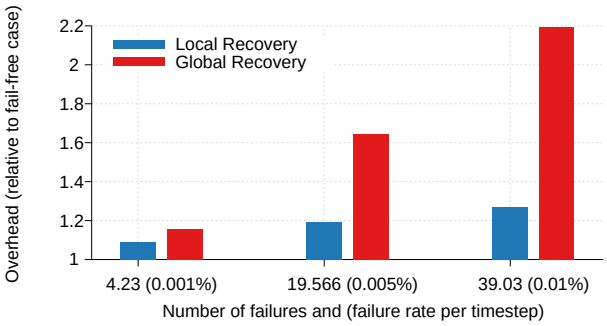


Figure 5: Recovery overheads for local and global recovery obtained from simulations based on the model for the parallel 3D stencil code running on a $16 \times 16 \times 16$ process grid.

from two failures sequentially. The likelihood of this situation is dependent on the communication pattern of the application and the checkpointing approach used. Specifically, it depends (1) on the dimension of the application domain (i.e. 1D, 2D, 3D), (2) on the size of the domain assigned to each process (which will determine the checkpoint latency), (3) the communication frequency, and, (4) the amount of computation per iteration (which will determine the latency between iterations, and is a factor of the size of the domain per node). In this work we explore an extreme scenario with a three-dimensional application domain, a relatively large data size, and communications every iteration.

Low power and energy footprint. Local recovery has better power and energy behavior as compared to global recovery as the entire system does not have to roll back and redo computations. Furthermore, in case of local recovery, while the neighboring processes wait for the re-spawned ones to catch up their CPU will be idle, and their power consumption can be reduced by using techniques such as Dynamic Voltage and Frequency Scaling (DVFS).

4. MODELING DELAY PROPAGATION

Section 3 covered the theoretical benefits of local recovery. This section goes one step further and tries to understand how these benefits can be mathematically modeled. Specifically, we construct a recurrence relation to simulate the execution pattern of a parallel stencil code and use it to quantify the impact of local recovery. Before moving on to the real evaluation in Section 5, this section will show how the presented equation can effectively model the behavior presented in Section 3.

4.1 Modeling the delay for a 1-D stencil code

As discussed in Section 3, in typical parallel stencil-based applications, every process communicates with its neighbors to exchange boundary data in each iteration. Based on this observation, we use the following recurrence relation to model the execution time of a 1-D stencil computation:

$$T(i,j) = \max(T(i-1,j-1), T(i-1,j+1)) + T_{local} + r$$
$$T(0,j) = 0$$
$$T_{local} = t_1 \text{ if no delay, or } t_2 \text{ if delayed}$$

where $T(i,j)$ is the total execution time at i-th timestep of j-th process. We assume that all the processes are assigned linearly to nodes so that every node communicates with its neighbors ($j-1$ and $j+1$). T_{local} is the execution time of the local computation per timestep, which is determined based on the probability of process failure; we define two cases, t_1 and t_2 to represent no-failure (no-delay) and failure (delayed), respectively. Additionally, a random performance noise r is applied in every timestep. The term, $\max(T(i-1,j-1), T(i-1,j+1))$, represents synchronization (communication) with neighboring processes; we assume that every process waits for the completion of its neighbors before starting its next timestep. This may not be true for state-of-art optimized applications, which are typically designed to compute on the interior mesh points while neighbor exchanges of ghost regions happen. Note that our model also does not consider the interconnect network performance and the checkpoint overhead. For example, the recovery from a failed process may involve allocating a spare process, which may dramatically change the network topology of the nodes around the failed ones. However, the goal of our model is to capture the execution pattern of stencil-based applications and to explore delay propagation and resulting performance impacts due to local recovery from failures. Note that as show in Section 5, results from this model do corroborate with experimental results. A more rigorous model and verifications would be helpful to better understand the relationship between system performance and application behavior and for optimizing our recovery schemes, and we plan to explore it in a part of our ongoing research.

4.2 Simulation

We have implemented a simulator using the model described above in Matlab and used it to study the progress of each process in the event of failures. The simulator computes the execution time at each core for every timestep using the recurrence presented above. The failure rates are provided as input parameters and are used to determine t_1 and t_2. These simulations do reproduce the progress of each timestep as presented in Figure 4. For example, the 32 core case shows how the first two delay propagations are merged. We observe similar results from the experiments using real 1D stencil code in Figures 8 and 9 presented in Section 5.8.

Next, we extended our model and simulator to model a 3D stencil (with 6 neighbors) on 4096 cores, to estimate the execution pattern of S3D on a 16^3 process grid for 100 timesteps. In this case, we use the set of parameters presented in the table below:

t_1	1.0	Failure rate (in %)	0.001, 0.005, 0.01
t_2	5.0	r	(0, 0.1)

We ran each simulation 30 times and plot the average execution time (for each core) for local recovery and global recovery in Figure 5. In case of the global recovery model, we apply the same delay factor t_2 to all the processes when a failure occurs in one (or many) of the processes. This may be optimistic as it does not consider the potentially high overhead for MPI communicator recovery, as reported by our previous work [16, 34], but we do apply the same delay effect to both cases for fair comparisons. The model-based simulation shows the delay from multiple failures is masked by local recovery, reducing the recovery overhead down to 22% of the global recovery case.

5. EVALUATION

Previous work [16] has shown how to tolerate failures in an online and global manner with failures occurring every 47 seconds. In this paper we have reduced the sources of overhead related to faults and, therefore, we want to experimentally evaluate how these algorithmic improvements affect the feasibility of injecting failures at even higher rates (i.e. every 5 seconds). Furthermore, we want to show that the local recovery algorithm implemented in Fenix can tolerate a set of failures and mask the total overhead so that it appears as if only a single failure occurred. In other words, we will empirically show that, with local recovery, the total overhead O_N of recovering from N failures is not necessarily $N \times O_1$ (being O_1 the overhead of recovering from a single failure).

In this section we will present the experimental evaluation of Fenix performed using the S3D combustion simulation on Cray XK7 Titan at ORNL.

5.1 Goal

The goal of the experimental evaluation is to demonstrate that using Fenix, even in tightly-coupled applications such as S3D, (i) node failures coming as frequent as every 5 seconds can be recovered (ii) failure recovery is scalable and (iii) the overhead of recovering from failures is not proportional to the system size.

All these make local recovery in Fenix a viable prototype towards exascale resilience for applications like S3D.

5.2 Methodology

We first present an scalability evaluation of the checkpointing technique implemented on Fenix using the S3D with a checkpoint size of 130 MB/core.

As a second step, we study the overheads related to the recovery process. To do it, we inject worst-case failures, i.e. sets of failures that do not allow recovery propagation delays to merge and, therefore, the total overhead is the sum of the recovery overhead for each failure. We show that Fenix handles frequent node failures up to MTBFs as low as 5-s with total overheads on or below 50% in the worst case scenario. This is an empirical demonstration that this method is more efficient than theoretical full redundancy for MTBFs greater than 5 seconds, in the targeted application type.

Finally, we show the full advantage of Fenix' local recovery capabilities on S3D up to 140736 cores (140608 + 128 spare cores). We demonstrate how in real scenarios local recovery can impose a total overhead of only one failure recovery, regardless the number of failures.

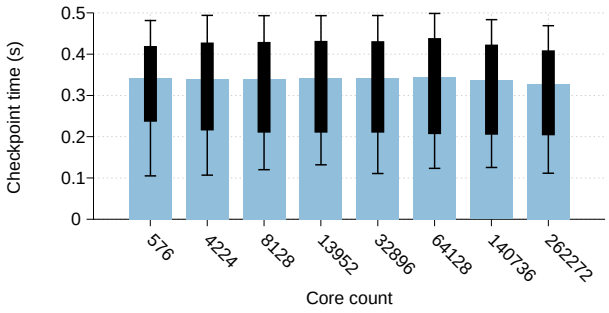


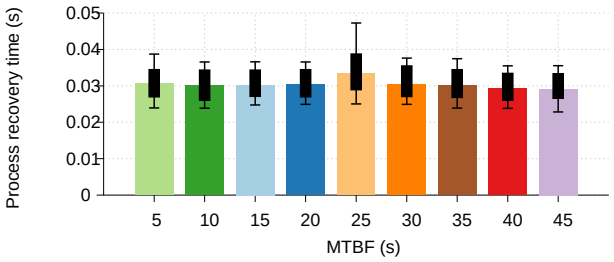
Figure 6: Checkpointing Scalability using 130MB/core.

In order to perform these experiments we inject node failures. Node failures are injected by simultaneously sending SIGKILL signals to all application processes running in that node. As the network setup parameters are stored in the process memory, by killing the process we do not allow any disconnection to be performed (which is what would happen if a real failure occurred). The processes on other nodes will get error codes when trying to perform a uGNI operation towards the killed processes, exactly as would happen if the node actually went down. In this sense, we say that we inject *real* failures, as opposed to just pretend that a process is dead. From now on, when we talk about failures, we refer to *node* failures, which is equivalent to *N-process* failures, being N the total number of processes of a node, two nodes, a blade, etc. By default, the experiments use $N = 16$.

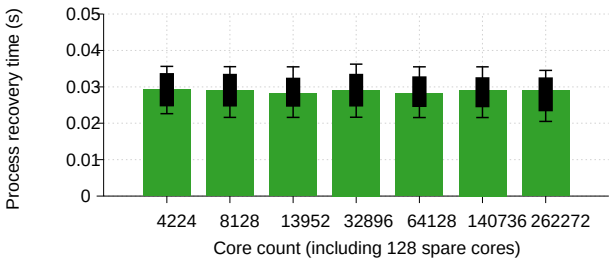
5.3 Implementation

Fenix has been designed in a modular and layered manner. Four main modules exist: data resiliency, process resiliency, communication logic and transport layer. The *communication* module is object oriented in order to isolate failure recovery mechanisms from the different communication protocols. A base class Command offers an abstraction for reliably requesting services or communicate with other processes using the chosen transport layer. All communication mechanisms (i.e. checkpoint, send/recv, barrier, broadcast, etc.) rely on this class and implement the logic in a simple manner without including any fault tolerance code. *Data resiliency* implements the checkpointing and data recovery mechanisms, which can be substituted by an interface to any existing checkpointing library. *Process resiliency* manages the spare process pool and implements the protocol to restart processes. The *transport layer* has been implemented using uGNI, but could be easily extended to other networks. Communications in production versions of MPI implementations are probably faster than with our framework, because we have not optimized performance of data transfer yet. Specifically, we cannot use RDMA in uGNI, due to inconsistent behavior of the network driver that appears only after injected failures. When a failure occurs, the core of the runtime invokes the different modules, orchestrates the recovery process at high level and determines which commands have to be re-executed.

Fenix offers three interfaces: C, C++ and Fortran. The programming overhead of using Fenix is low, requiring less than 35 new, changed, or rearranged lines of code in S3D,



(a) Different MTBFs. Core count fixed at 4736 cores (4096 compute cores and 640 spare cores).



(b) Different core count. MTBF fixed at 10 seconds.

Figure 7: Average process recovery overhead.

as shown in Section IV.B on [16]. Indeed, the programmability of Fenix for local recovery is even simpler than the one required by global recovery, because the status returned by `Fenix_Init()` can take three different values for global recovery (i.e. New, Respawned, or Survivor) and only two values for local recovery (i.e. Survivor status is no longer valid, since all survivor processes don't get interrupted from their processing). This simplifies the application side of the recovery logic.

We leave the detailed description of the architectural design of Fenix and its implementation for future work.

5.4 Testbed

As already mentioned, all the experiments in this paper were done on the Cray XK7 Titan at ORNL. Titan is composed of 18688 16-core CPU and the same number of GPUs. Every pair of nodes is connected to a single custom system-on-chip Gemini ASIC network interconnect. Gemini ASICs are connected using a 3D torus topology. Applications can directly access network capabilities using uGNI, the user level proprietary interface from Cray, which is forward compatible with newer versions of Cray networks, such as Aries.

5.5 Checkpoint Scalability

In [16] double in-memory checkpoints were evaluated and proved to scale independently of the number of processes. With this newer version of the Fenix framework we reevaluated the checkpoint scalability even when injecting failures, and with data sizes per core up to 16 times larger. Figure 6 depicts the weak scalability of checkpointing up to 262272 cores (including 128 spare cores) using a larger checkpoint size of 130 MB/core. The test was done injecting failures every 10 seconds. As we can see, the checkpoint overhead is the same independently of the number of cores, which demonstrates ideal weak scalability.

5.6 Recovery time for different MTBFs

The goal of this second experiment is to show that our implementation is capable of handling failures occurring up to every 5 seconds. We will also study process recovery overhead and empirically demonstrate that its performance does not depend on the total number of processes (i.e. optimal scalability), which is a highly desirable property towards exascale. We performed the experiment using S3D and 4736 cores (4096 compute cores and 640 spare cores), unless otherwise specified.

In this experiments we engineer the set of failures in a way that do not allow recovery propagation delays to merge and, therefore, the total overhead is the sum of the recovery overhead for each failure. In other words, this is a worst-case local recovery test, because failures are injected so that the recovery overhead of different failures is not merged together. This is done in order to explore the benefits of the local recovery algorithm over global recovery and the potential benefit for all kinds of applications, not just stencil-like architectures.

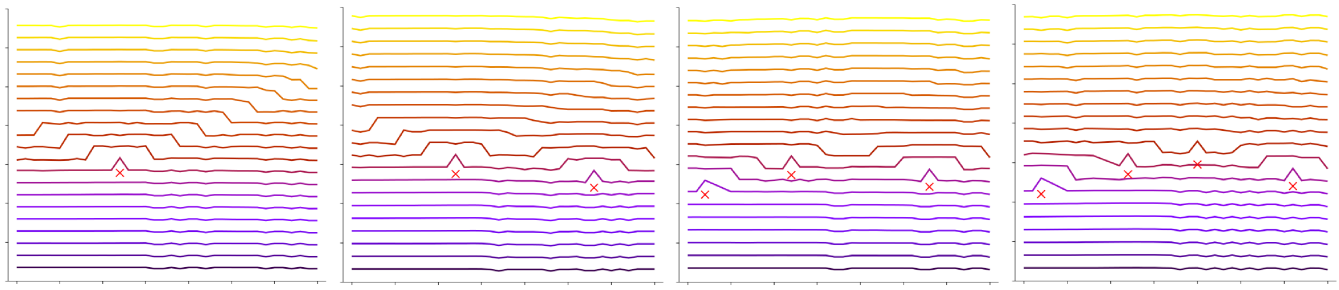
Process recovery overhead. First, we will study the total overhead to recover from a failure for different failure frequencies. Figure 7(a) shows the average overhead of process recovery for different node failure frequencies injected on S3D on an execution of about 200 seconds. Each bar represents the average time to recover the processes from a failure. This overhead is only initially seen in the spare processes that substitute the failed ones, and then propagated to the rest of the domain due to the communication nature of the application. This figure does not include overheads due to data recovery (i.e. checkpoint fetching) Note that checkpoint fetching is exactly the inverse process of checkpointing: first, checkpoint has to be fetched from the neighbor that stores it in-memory to the spare process memory. Then, it has to be `memcpy()`'ed to the application store. Therefore, checkpoint fetching overhead must be very similar to checkpointing overhead.

Scalability. Figure 7(b) shows the average process recovery time for every failure. The figure demonstrates that local recovery overhead is constant regardless the number of processes in the system, tested up to 262272 cores. The test was done with a fixed MTBF of 10 seconds and all the tests include 128 spare processes. Most of the tests (excluding 64k and larger) have been repeated with different total number of failures (e.g. ranging from 1 to 8 failures), and the results have been averaged.

5.7 Total overhead for different MTBFs

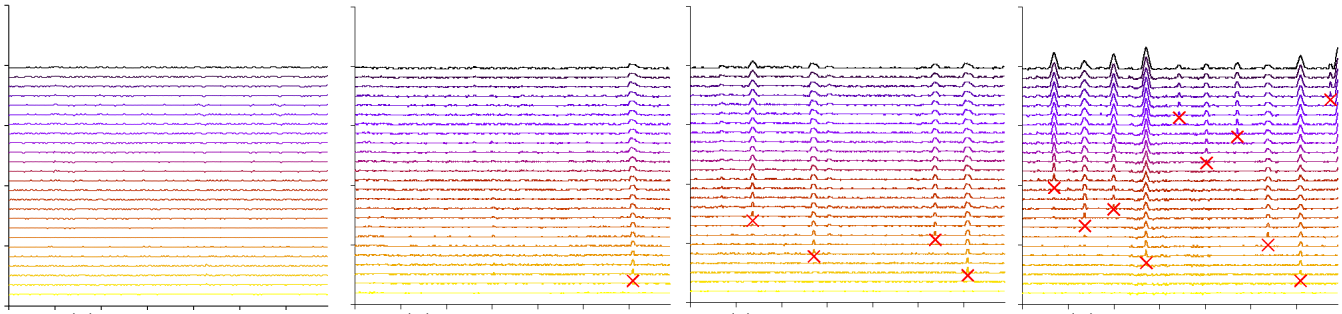
Once we understand that process recovery time from a single failure is constant (and small) independently of the total number of cores in the system and the failure frequency, we are ready to study the total overhead due to fault tolerance (i.e. including checkpointing, process/data recovery and rollback overheads). In other words, we are interested in comparing the end-to-end execution time of a failure-free, checkpoint-free, execution with the end-to-end execution time of different executions at different MTBFs.

Figure 10 shows the results of the experiment, which has been executed using a fixed core count of 4096 cores and 640 spare processes and a checkpoint size of 53 MB/core. For different failure rates ranging from 5 to 45 seconds, Figure 10 shows the total overhead relative to a failure-free checkpoint-free base test. The total number of failures ranges from 48



(a) One process failure (b) Two process failures (c) Three process failures (d) Four process failures

Figure 8: Behavior of local recovery on 1D PDE using 36 cores (32 compute cores and 4 spare cores).



(a) Base execution (b) One node failure (c) Four node failures (d) Ten node failures

Figure 9: Behavior of local recovery on 1D PDE using 13984 cores (13824 compute cores and 160 spare cores), with failures inserted every 10 seconds.

processes (3 nodes) to 528 processes (33 nodes), as indicated on top of each bar during a total time of around 150 seconds.

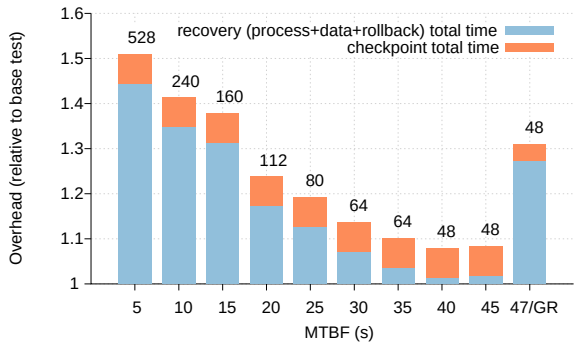


Figure 10: Comparison of total overhead due to failures and fault tolerance mechanisms with different MTBFs. Core count fixed at 4736 cores (4096 cores and 640 spare ranks). Checkpoint size is 53MB/core. Numbers on top of the bars indicate the total number of process failures injected during the execution.

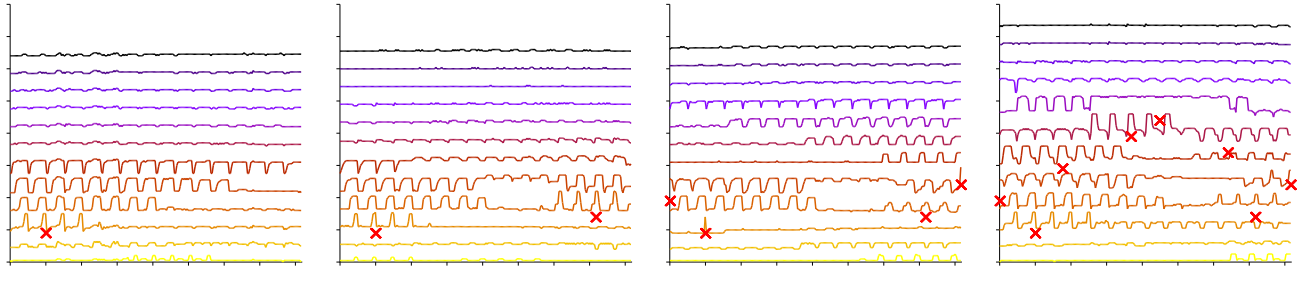
The right-most bar in the figure shows the overhead of the same test using global recovery while injecting failures every 47 seconds, as described in [16]. We can see that using local recovery we obtain much lower performance penalties even at higher failure rates. For example, notice that local recovery allows failures every 20 seconds with an overhead below 25% while global recovery allows failures every 47 seconds (∼2.35x) with an overhead of 31% (∼1.25x). Note that the total overhead of running the experiment in a worst-case scenario (i.e. having node failures arriving every 5 seconds), is 51%. That is, it is slightly worst (1% worst) than the theoretical best-case overhead of using 2-way redundancy. Again, it is important to note that this experiment has been done injecting worst-case failures, in which failure masking does *not* occur in most cases (note that the exception is the 5-s MTBF test in which the total time is just a small portion above the 10-s MTBF test). This has been done to study only one of the advantages and scalability of process recovery. Next subsection studies the full advantage of local recovery, but the best case (i.e. where the overheads from all failures merge in just one) from the current experiment can be obtained by dividing the recovery overhead in Figure 10 by the number of node failures.

5.8 Local Recovery and Failure Masking

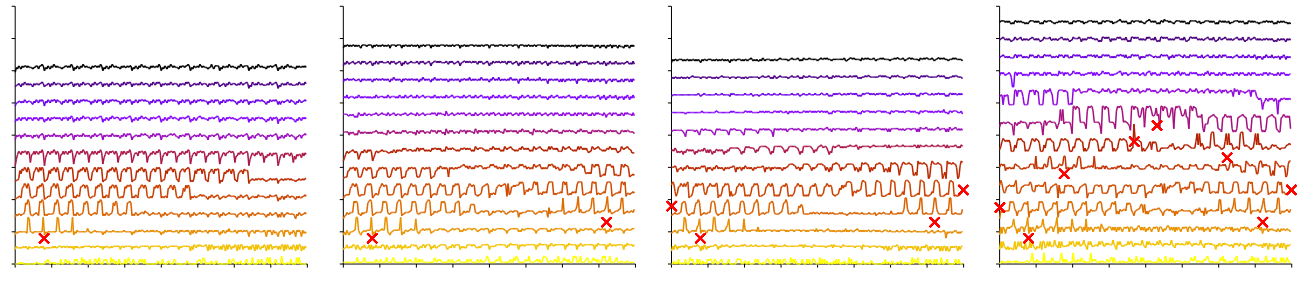
The goal of this last experiment is to show that, with local recovery, the total overhead O_N of recovering from N failures is not necessarily $N \times O_1$ (being O_1 the overhead of recovering from a single failure). This is what we call failure masking.

1D PDE. We will begin by studying the behavior of local recovery in a Stencil-structured unidimensional Partial Differential Equation solver, or 1D PDE for short.

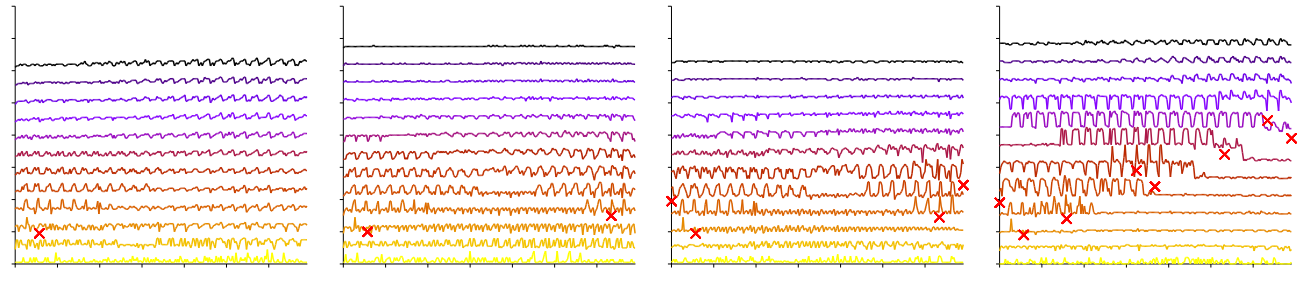
Figure 8 shows several executions using 32 compute cores and 4 spare cores. The results were obtained via real execution on Titan Cray XK7 at ORNL, in which we injected real process failures. The X axis represents the core number (or rank number), while the Y axis represents wall time clock. Each rank simulates a certain number of points in the



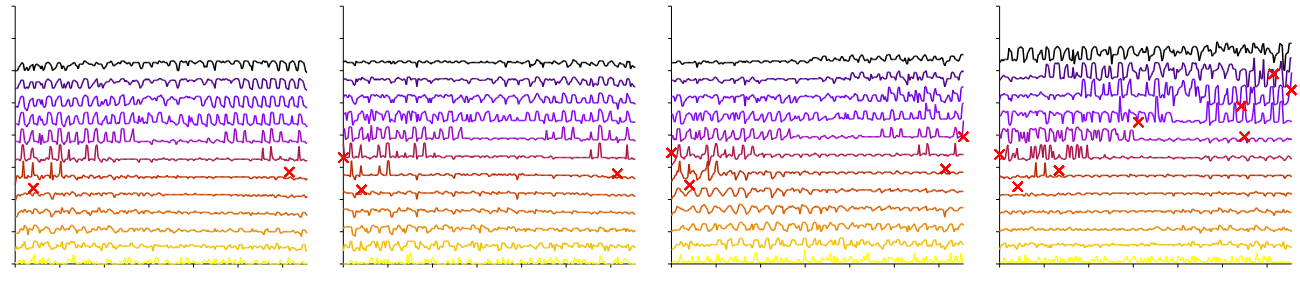
(a) 4224c 1f (b) 4224c 2f (c) 4224c 4f (d) 4224c 8f



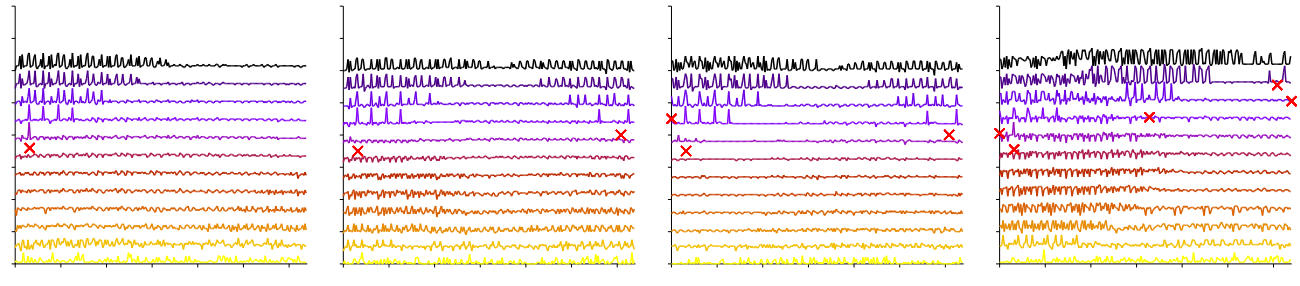
(e) 8128c 1f (f) 8128c 2f (g) 8128c 4f (h) 8128c 8f



(i) 13952c 1f (j) 13952c 2f (k) 13952c 4f (l) 13952c 8f



(m) 32896c 2f (n) 32896c 3f (o) 32896c 4f (p) 32896c 8f



(q) 64128c 1f (r) 64128c 2f (s) 64128c 3f (t) 64128c 5f

Figure 11: Behavior of local recovery in S3D Stencil three-dimensional Partial Differential Equation solver. Each line in a figure represents a timestep, and the color of the line represents how advanced the simulation is (i.e. it advances from yellow to dark purple). Each red cross represents a failure. A straight line means all processes compute the timestep at the same physical time (note how in some cases performance fluctuation create wavy lines. When a failure occurs, the recovery delay does not get propagated immediately to all the domain. On the contrary, the immediately adjacent neighbors in the 3D space are the first to be delayed, which in turn propagate the delay to their contiguous neighbors in subsequent iterations, eventually reaching the whole domain. The X axis of the figures shows the rank number, which is mapped to the 3D domain linearly. The Y axis of the figures shows the wall time. In the caption of each figure a 'c' refers to cores and an 'f' refers to the number of node failures injected. Note how the overhead in the first three columns is the same, which empirically demonstrates how failure masking works.

domain, which is a 1D space (i.e. line). Adjacent ranks simulate adjacent parts of the line. Each horizontal line in a plot represents a simulated timestep (i.e. every time the solver communicates with the neighbors in order to advance the simulation). Red crosses represents injected process failures. A straight line means all processes compute the timestep at the same physical time. When a failure occurs, the recovery delay does not get propagated immediately to all the domain. On the contrary, the immediately adjacent neighbor process is the first to be delayed, which in turn propagate the delay to their contiguous neighbors, eventually reaching the whole domain. Note how Figures 8(b), 8(c) and 8(d) have the same recovery overhead: as if only one failure occurred, Figure 8(a).

Figure 9 shows a longer experiment using a larger number of processes and injecting node failures: 13984 total cores, including 13824 compute cores and 160 spare cores. As we can observe, in this case the propagation waves occasioned by one failure recovery never merge with other waves. Therefore, the overhead in the total time to solution will be only increased by the time to recover from a single failure, independently of the total number of failures.

By comparing Figure 8 with Figure 4 (left) and Figure 9 with Figure 4 (right) we can see that results from the model and simulation accurately predict the real experimental results. We can conclude, therefore, that the presented discrete event simulator can capture the benefit of local recovery. These results not only validate our algorithm and implementation, but also show that local recovery can be benefitial in extreme scale environments, where failures are predicted to appear at high frequency. Local recovery is a scalable approach in both the number of failures and the size of the machine.

S3D. Figure 11 shows the profile of multiple S3D executions. Figures follow the same format as Figure 8. Here, however, the communication pattern between ranks is not as obvious as with the 1D case and, therefore, the delay will be propagated in a strange pattern. The communication in the simulated 3D domain goes as follows: each process communicates with six other processes, responsible of simulating points in the neighboring up/down, front/back and left/right points. The mapping between 3D space and 1D ranks is done in a straightforward manner. By beginning in the point (0,0,0) we assign rank numbers by counting first in the Z direction, followed by the Y direction and finally in the X direction.

For each scale (576, 4224, 8128, 13952, 32896, 64128 and 140736 cores), different number of failures has been injected, from 1 to 8. We show that the total overhead is, in most cases, as if only one failure occurred. This can be seen in a summarized way in Figure 12. This figure shows the relative execution time compared to the base test of just recovering from one single failure. Note how, in many of the cases, the execution time difference is around 2% or below, which indicates similar execution times. The difference might be due to different rollback overheads: if failures occur right after a checkpoint is done, the rollback overhead is small meanwhile if a failure occurs right before a checkpoint should be done, the rollback overhead is much larger, because almost an entire iteration has to be recomputed. The tests that take larger execution times, up to 6 or 12% compared to injecting a single failure, cannot be considered the same overhead, which means that there was at least one failure

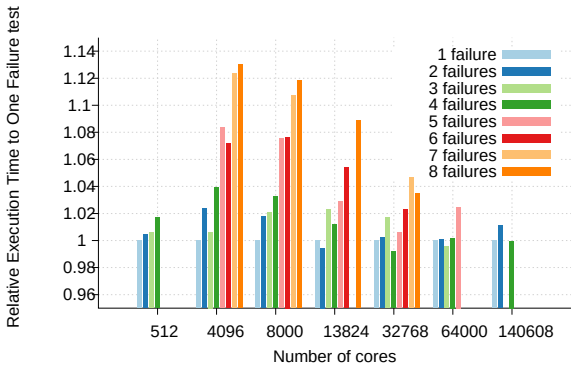


Figure 12: Summary of relative total execution time of injecting multiple failures, compared to the execution time while injecting a single failure. For 64k and 140k cores, only the test that went through Titan's execution queue are shown. For 262k cores only one test went through the execution queue, so no comparison is available.

that occurred within the propagation delay of the recovery of a previous failure. This is consistent with Figure 11. For example, if we compare Figure 11(d) with the rest of the tests using 4224 cores, we can observe a large difference in the total overhead. This difference is due to a failure occurring in a core which already suffered delay from a previous failure in a neighboring core. In this case, the third failure (counting from the left) suffers from the effects of the second failure.

We have shown how local recovery enables failure masking, which is a highly desirable property towards extreme-scale systems. We have empirically demonstrated that, with local recovery, the total overhead O_N of recovering from N failures is not necessarily $N \times O_1$ (being O_1 the overhead of recovering from a single failure).

6. CONCLUSION

In this paper we have explored an approach for failure masking and online, local recovery from high-frequency node failures for stencil-based parallel applications, using application guided checkpointing as a means for data resilience. The approach is based on understanding the propagation of delays associated with local recovery for stencil-based computation/communication pattern, and the observation that the impact of the delay associated with multiple failures is often not additive, allowing this approach to be feasible and scalable. In this paper we first used simulations to validate the approach and then design and implement it within the Fenix framework. We then deployed it on the Titan Cray XK7 production system at ORNL and used it to enable failure masking and local recovery for the S3D combustion application, which is a key application for exascale [2] – the simple Fenix local recovery interface enabled this integration to be achieved with less than 35 lines of S3D code to be added, changed, or rearranged.

We also presented an extensive experimental evaluation of the local recovery algorithms in Fenix on Titan using S3D, while injecting real failures. Our experiments demonstrate that local recovery in Fenix provides a viable solution for stencil-based application, for addressing node failures occurring as frequently as every 5 seconds, on scales up to

262144 cores. Our results show an overall performance overhead of 13.75% while tolerating node failures every 30 seconds (worst-case scenario, as the MTBF for the exascale systems is expected in order of minutes [13]). We also show that this overhead is smaller as compared to the overheads of using global recovery techniques previously presented on Fenix [16], and is also smaller than the overhead of traditional checkpointing with global offline restart that is currently used by S3D for its large-scale production runs on Titan.

Our experiments also evaluated the scalability of the local recovery stages in Fenix, and demonstrate that all fault tolerance aspects scale well with increasing number of cores. Finally, our experiments also demonstrated how local recovery enables failure masking, i.e., the overheads on the total execution time due to recovery from multiple failures is comparable to that of only one failure.

Our ongoing and future work includes (1) exploring how the conclusions in this paper apply to other classes of applications (beyond stencil-based), (2) modeling and understanding optimal sizes for ghost regions so that delays due to local recover propagates slower, and (3) studying how asynchronous transfers of checkpoints can reduce checkpointing overheads without impacting simulation time.

Acknowledgments

The authors would like to thank Josep Gamell, Robert Clay and George Bosilca for interesting discussions related to this work. The research was conducted as part of RDI^2 at Rutgers University. It was supported by the NSF via grant number ACI 1339036, by the DoE RSVP grant via subcontract number 4000126989 from UT Battelle, and used resources of the OLCF at the ORNL. This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. 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.

7. REFERENCES

[1] S. Amarasinghe and et al. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, DARPA IPTO, Air Force Reserach Lab, Sept. 2009.

[2] S. Amarasinghe and et al. Exascale Programming Challenges. In *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Jul 2011.

[3] 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*, SC 2011, 2011.

[4] 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. Technical report, US Department of Energy, December 2012.

[5] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. A Proposal for User-Level Failure Mitigation in the MPI-3 Standard. Technical report, Innovative Computing Laboratory, University of Tennessee, February 2012.

[6] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 2013.

[7] 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*, 20(3):319–333, 2006.

[8] 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*, pages 1–9, 2009.

[9] B. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello. Coordinated checkpoint versus message log for fault tolerant MPI. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 242–250, 2003.

[10] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[11] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2(1):015001, Jan. 2009.

[12] 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*, pages 18–18, Nov 2006.

[13] J. Dongarra and et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[14] 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.*, 34(3):375–408, Sep 2002.

[15] 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*, SC 2012, pages 78:1–78:12, 2012.

[16] 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*, SC '14, 2014.

[17] 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)*, pages 989–1000, 2011.

[18] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[19] M. A. Heroux. Toward Resilient Algorithms and Applications. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, FTXS 2013, pages 1–2, New York, NY, USA, 2013. ACM.

[20] J. Hursey. *Coordinated checkpoint/restart process fault tolerance for mpi applications on hpc systems*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2010. AAI3423687.

[21] 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*, HPDC 2009, pages 49–58, New York, NY, USA, 2009. ACM.

[22] 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*, pages 1–8, 2007.

[23] 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)*, pages 148–157, 2012.

[24] 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*, SC 2012, pages 1–11, Nov 2012.

[25] 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*, pages 29–40, May 2013.

[26] D. S. Katz, J. Daly, N. DeBardeleben, M. Elnozahy, B. Kramer, L. Lathrop, N. Nystrom, K. Milfeld, S. Sanielevici, S. Cott, and L. Votta. Fault Tolerance for Extreme-Scale Computing Workshop, Albuquerque, NM - March 19-20, 2009. Technical Report ANL/MCS-TM-312, Argonne National Laboratory, December 2009.

[27] 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*, SC 2010, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[28] 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*, SNAPI 2010, pages 13–20, Washington, DC, USA, 2010. IEEE Computer Society.

[29] 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)*, pages 375–384. IEEE, 2011.

[30] 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*, HPDC 2013, pages 143–154, New York, NY, USA, 2013. ACM.

[31] 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*, SC 2013, pages 8:1–8:12, New York, NY, USA, 2013. ACM.

[32] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. d. Supinski, N. Maruyama, and S. Matsuoka. Fmi: Fault tolerant messaging interface for fast and transparent recovery. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1225–1234, Washington, DC, USA, 2014. IEEE Computer Society.

[33] 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.

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

[35] M. Turmon, R. Granat, D. Katz, and J. Lou. Tests and tolerances for high-performance software-implemented fault detection. *IEEE Transactions on Computers*, 52(5):579–591, 2003.

[36] 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)*, pages 1–6, 2012.