APPLICATION-AWARE ON-LINE FAILURE RECOVERY FOR EXTREME-SCALE HPC ENVIRONMENTS

By

MARC GAMELL BALMANA

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Manish Parashar

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2017

ABSTRACT OF THE DISSERTATION

Application-aware On-line Failure Recovery for Extreme-scale HPC Environments

By MARC GAMELL BALMANA

Dissertation Director:
Manish Parashar

High Performance Computing (HPC) brings with it the promise of deeper insight into complex phenomena through the execution of various extreme-scale applications, especially those in the fields of science and engineering. The increasing computational demands of these applications continue to push the limits of current extreme scale HPC systems. As a result, the community is working toward achieving exascale systems able to compute 10^{18} floating point operations per second (FLOPS). Since these systems are expected to contain a large number of components, reliability is one of the key anticipated challenges. Due to the extensive periods of time that complex applications require, future systems will likely see an increase in process and node failures during application execution. These failures, also known as hard failures, are currently handled by terminating the execution and restarting it from the last stored checkpoint. This checkpoint-restart methodology requires the application to periodically save its distributed state into a centralized, stable storage –an approach that is not expected to scale to future extreme-scale systems. While the illusion of a failure-free machine –implemented either via hardware or system software strategies– is adequate for current HPC systems, they may prove too costly in future extreme-scale machines. Resilience is, therefore, a key challenge that must be addressed in order to realize the exascale vision.

This dissertation explores new models that leverage application-awareness to enable on-line failure recovery. On-line recovery, which does not require the interruption of surviving processes in order to collectively restart the entire application, offers better cost/performance tradeoffs by reducing recovery overheads. Recovering processes on-line enables application-specific data recovery strategies and optimized in-memory checkpointing while avoiding the repetition of initialization procedures –the least optimized part of most production-level applications– on all processes.

This dissertation addresses three areas of research in on-line failure recovery. First, it explores a generic global on-line recovery model, involving all processes in the recovery process. Second, it explores optimized local recovery in which communication characteristics of certain application classes are leveraged to reduce overheads due to failure. In particular, finite difference partial differential equation solvers using stencil operators are used as the driving application class. Third, this dissertation demonstrates how the overhead of multiple, independent failures can be masked to effectively reduce the impact on total execution time. The models presented in this dissertation are implemented and evaluated in Fenix and FenixLR, a pair of generic and extensible frameworks used to demonstrate the concepts.

Acknowledgments

First, I would like to thank my dissertation advisor Dr. Manish Parashar for his invaluable advice, guidance, support, and example. His optimism, energy, and enthusiasm have helped and motivated me throughout my studies at Rutgers.

I would like to thank as well Dr. Ivan Marsic, Dr. Deborah Silver, and Dr. Keita Teranishi for serving as part of my committee members.

I thank Dr. Keita Teranishi for continuously supporting Fenix as well as my research, for his collaboration, productive discussions, and ideas, and for providing me with the opportunity to join his team as an intern.

I would also like to thank Dr. Rob Van der Wijngaart for his constructive criticisms and support of the Fenix framework as well as his invaluable help while formally specifying the Fenix interface.

I thank Dr. Daniel Katz, Dr. Hemanth Kolla, Dr. Jacqueline Chen, Dr. Jackson Mayo, Dr. Janine Bennett, and Dr. Scott Klasky for their collaborations, valuable discussions, and access to leadership HPC systems.

I wish to thank all of my colleagues, Dr. Ivan Rodero particularly, at The Applied Software Systems Laboratory ($TASSL$) and Rutgers Discovery Informatics Institute (RDI^2) for their cheerfulness and help during my years at Rutgers.

Last but not least, I would like to express my deepest gratitude to my parents, grandparents, the rest of my family, and Bethann for their support, love, and inspiration. I would like to specially thank my father, Josep, for his guidance, technical advice, and help in all the stages of my studies. My parents have been by my side in every step and never stopped encouraging me.

Dedication

To my parents Josep and Gemma, and to Bethann.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Exascale. To satisfy the increasing demands of science and engineering applications, the computational power of extreme-scale systems must be increased. As a result, the HPC community is committed to achieving working systems able to perform 10^{18} floating point operations per second (FLOPS) by the end this decade [49, 48, 3] or early in the next decade. Governments have also recognized the need to advance science through the use of high performance systems. For example, in 2015, U.S. President Barack Obama issued an executive order [119] urging the community to accelerate the delivery of an exascale system over the next decade.

While exascale systems will enable computations at unprecedented scales and resolutions, ultimately leading to dramatic insights into complex phenomena, achieving this goal presents significant challenges. Current petascale systems have millions of cores (e.g., the top system on the Top500 list, as of November 2016, achieves 93+ petaflops with 10,649,600 cores [147]) and next-generation exascale systems are expected to have an order of magnitude more. To achieve such computational power, it is projected that individual compute nodes will contain thousands of cores [145]. Due to the anticipated, very large number of cores and components, the DoE ExaOSR report [7] identifies an intensified focus on resilience as one of four key challenges toward exascale along with the containment of power consumption, enhanced memory access, and a dramatic increase of parallelism. Emphasis on these challenges has been further documented in other reports such as the Blue Waters & TeraGrid 2009 workshop [96], as well as the Argonne National Laboratory's 2012 resilience workshop report [139].

Low reliability at scale. The mean time between failures (MTBF) for current petascale-level systems (i.e., systems with a compute power in the order of 10^{15} FLOPS) is measured in hours, and the failures typically affect either a single node or a small number of nodes. Studies [146] show that failures, even when they are independent and therefore not related, present statistical temporal locality. In high-end systems, this translates to long periods of *high stability* (no failures) separated by periods of *high instability* (failures occurring much more frequently than the MTBF). It has been shown that stable periods can last as much as three times longer that the MTBF while failures occurring during unreliable periods may occur, for example, more than seven times as frequently as the MTBF. This can be observed through executions performed on the Titan Cray XK7 at ORNL OLCF, ranked third on the Top500 list as of November 2016[1] [147]. According to Tiwari et al. [146], about 29% of the failures observed on Titan over the course of several years occur within an hour of the last failure while about 45% occur within three hours of the last failure despite Titan's MTBF of approximately 7.75 hours. One particular production run using 130,000 of Titan's cores (which represents 8,125 compute nodes, or 44% of Titan's total compute nodes) experienced nine node failures during a 24-hour period, indicating a period of *high instability* in the machine's lifetime.

A dramatic improvement in component reliability is required to neutralize the effect of the aforementioned heightened component counts that future extreme scale machines will require. Architectural trends such as the predicted reduction of gate width and voltage levels [139], however, suggest that component reliability may, in fact, decrease. As a result, while it is difficult to predict what the actual MTBF of an exascale machine will be, some researchers believe that it might be in the order of minutes [49]. This dissertation does not assume any particular predicted MTBF, but instead approaches the problem using worst-case analysis.

Risks and costs of increased hardware-level resilience. Implementing fault tolerance directly at the hardware level has clear advantages such as allowing the software to build on a reliable substrate. Offering an abstraction of a failure-free machine, however,

[1]Titan was ranked as the first system in the November 2012 list and as the second system from June 2013 to November 2015.

can have significant costs associated. First, there is the cost of developing hardware-level resilience techniques and the per-component cost to implement these techniques in silicon. Second, hardware-based fault-tolerance can incur additional operating costs, such as increased power requirements, in order to keep the extra hardware components running. An example can be found in the hardware-implemented Error Correcting Code (ECC) technology that is shipped with high-end DRAM modules. ECC consumes a certain amount of power to function which can, in turn, add up to a non-negligible percentage of the power consumption of an HPC center. Increasing power consumption should be avoided as much as possible, as keeping a tight power budget is another big challenge that needs to be addressed in order to achieve exascale [7]. Since an exascale machine is expected to run on 20 MW, the HPC community is seeking a machine that will be $1,000,000/93,014.6 \approx 11\times$ more powerful than the most powerful machine (as of November 2016[2]), but will only be consuming $20,000/15,371 \approx 1.30\times$ more power. In order to stay within the target power limit, one school of thought projects that only a small part of each chip will be powered at a certain time so that most of the chip will not be operational most of the time (an effect called dark silicon [56]). If chips contain power-hungry hardware dedicated to tolerate failures, machines will be required to turn off other parts of the hardware in order to turn on resilience features, effectively trading off performance to improve resilience.

Heroux [80] forecasts that, while scientists and engineers do not develop mechanisms to run their simulations in a failure-prone scenario, HPC centers and vendors will keep building reliable machines regardless of performance penalties imposed by hardware-level fault tolerance. Only when the community is ready will the market provide less reliable, higher performing HPC systems. It is, therefore, imperative to provide efficient support for failure resilience that presents more attractive tradeoffs than today's mechanisms.

1.2 State of the Art Software for Hard Failure Resilience

It is likely that future machines will experience a significant reduction in MTBF which may be unavoidable or, as discussed above, voluntary and accepted by applications. This MTBF

[2]According to Top500, Sunway TaihuLight [147].

reduction will directly impact applications since the typical run time of target scientific and engineering applications will be longer than the MTBF. For this reason, resilience will be a key design requirement for exascale systems and applications and fault tolerance techniques will become essential. An important class of failures that must be addressed is process and node failures (also known as hard failures or fail-stop failures), including the correlation effects in which a set of failures is triggered by a previous one. The mechanisms implemented in this dissertation aim at addressing hard failures since other types of failures could, potentially, be promoted to fail-stop failures when detected. The presented mechanisms could, therefore, be used to tolerate other types of failures.

Research on runtime and OS level resilience explores mechanisms that offer a view of a failure-free machine to the application by transparently handling failures. While these techniques are attractive due to their simplicity, they offer applications only generic solutions, often at high costs. Consequently, current production-level HPC systems do not provide fault tolerant runtimes out-of-the-box. For example, hard failures that occur during the execution of an application using the Message Passing Interface (MPI) are considered fatal by default and result in all application processes aborting. As a result, process and node failures are often recovered using *offline* techniques that restart the job or the MPI environment from scratch, usually restoring 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 these failures, it is unclear whether this approach will scale to exascale since the time to checkpoint will often be longer than the expected MTBF. An important body of works actively focuses on C/R improvements to address these issues [112].

As an alternative approach, several fault tolerant MPI runtimes have been developed, such as MPICH-V [20], rMPI [60], MR-MPI [55], redMPI [64], and FEMPI [141]. These runtimes aim to support completely transparent failure recovery (i.e., they offer an abstraction of a fault-free system to the application). To do so, they use one or more of the existing, application-agnostic fault tolerance techniques (e.g., creating coordinated checkpoints, logging messages, or replicating computational resources).

Need for application-aware resilience. However, the abstraction of a failure-free software stack comes with high overheads, as happens with hardware-level fault tolerance. As suggested by recent studies [79], this abstraction will not be sustainable at extreme scales due to the costs of implementing resilience techniques in the hardware, OS, or runtime levels. Therefore, application-aware resilience techniques will likely be required at exascale.

1.3 Research Challenges for Application-assisted Resilience

1.3.1 Support for Customizable Application Resilience

Traditional single program multiple data (SPMD) and message passing programming models were not designed to tolerate failures. As the need for resilience becomes more prevalent, a large number of techniques have emerged to enable resilience in applications using these paradigms.

Application-agnostic techniques, such as automatic checkpoint and restart or redundancy, focus on hiding the failures from the application. However, they result in high levels of overhead and, therefore, are rarely used in production.

Alternatively, some application-aware techniques, such as Algorithm-Based Fault Tolerance (ABFT), focus on soft failures and are mainly designed for specific types of applications. For example, ABFT strategies focus on operations with matrices (e.g., linear algebra operations such as matrix-matrix multiplication).

Other application-aware techniques, such as application-based checkpoint and restart, can address the high overhead of runtime-based checkpoint and restart by selecting only the parts of the data that need to be saved. Even though some SPMD optimizations store checkpoints in memory, they only do it as a caching mechanism before pushing them to a centralized parallel file system (PFS). When a failure occurs, in order to restart from a set of checkpoints, all ranks are required to reload from the most recent checkpoint found in centralized resources, causing contention. This is suboptimal since the great majority of ranks could still have a valid checkpoint stored in local memory and, therefore, achieve a faster restart.

Therefore, there is a need for a generic, customizable framework that enables production codes to be independent of traditional PFS-based checkpointing with offline restart. If application-specific/domain-specific resilience algorithms are not used, this framework must allow process and node failures to be recovered by leveraging generic, optimized, application-aware resilience mechanisms.

1.3.2 Support for On-line Recovery

Production-level applications are rarely monolithic. Instead, they are more often composed of multiple components and libraries used in conjunction to perform complex calculations. Since the typical run time of these applications can often be measured in the order of tens of hours or even days, developers focus their efforts on optimizing the parts of the code that account for large portions of the execution time; e.g., the operations in the critical path of the main loop. Therefore, other parts of the execution, such as the initialization of the application itself, its components, and the libraries that are used, are generally not optimized and can be very costly.

When a failure strikes applications that use traditional recovery procedures, all SPMD ranks (or processes) are shut down and the message-passing domain is re-started from scratch, launching all ranks from the beginning. These ranks are then required to re-initialize and, most likely, recover a previously saved global checkpoint from stable storage.

Current techniques prevent optimized **process and environment recovery** because modules and libraries that were initialized in all survivor ranks need to be restarted. Since all application data is lost when the ranks or processes are shut down, current techniques also prevent optimized **application-specific data recovery** that does not require data replication. For example, some applications could tolerate an approximation on the lost data, inaccessible due to the loss of a process or node, without the need to checkpoint –e.g., stencil applications can tolerate the loss of subdomains since these can be approximated by using the contents of logically neighboring subdomains, still stored in the memory of survivor processes. Another example can be found in distributed numerical methods which iteratively try to minimize a parameter, the error, to reach convergence. In some cases, the

loss of memory contents of a process or a node may not prevent reaching convergence, but, instead, may only imply the need to compute extra iterations before reaching it.

The required solution cannot therefore power off or disrupt survivor ranks and their memory contents when a failure is detected; there is a need of a runtime able to allow these ranks to continue running despite the failure.

1.3.3 Support for Existing Code Base

Some programming models offer a natural, flexible, and mostly automatic way to recover from failures. For example, applications that follow a task-centric model decompose the different computational parts into separate components, or 'tasks', which are typically fine-grained. While this model has its own set of challenges and overheads related to fault tolerance, its resilience can be managed in an automatic manner by rescheduling tasks that were assigned to a failed node.

Some applications and simulations, however, can easily divide their computational components statically among the different processors in an SPMD fashion. In these cases, the inherent overhead that comes from task-centric's dynamism and flexibility may, in fact, decrease performance when compared to an static, SPMD-like division of work. Nevertheless, SPMD and message passing models are not designed to handle process and node failures by default.

Furthermore, from a practical perspective, most production-level applications and simulations contain hundreds of thousands of lines of code that have already been written following an SPMD and message passing model and have been highly optimized for several platforms and situations over the decades. Trying to port these applications to a task-centric model would require enormous amounts of research, development, and optimization, which makes this option infeasible.

The de-facto standard for message passing applications, the MPI standard, does not have support for fault tolerance as of its third version. The User-Level Failure Mitigation (ULFM) proposal aims at adding the minimal set of operations to the MPI standard in order to support fault tolerance. This proposal's goal is to maximize flexibility so that the maximum number of recovery mechanisms can be built on top of it. This comes at the

expense of readability and ease of use. By definition, this flexibility comes with the cost of a highly invasive nature when it comes to the original application code, making it infeasible to use in large code bases. For this reason, the ULFM proposal is specifically crafted to be used as the basis for fault tolerance libraries and may not be intended to be used directly by the application code.

A solution for process and node failures that does not require the re-write of hundreds of thousands of lines, including complex optimization procedures, is still, therefore, needed. The solution needs, instead, to leverage the research, development, and performance optimizations achieved during the last few decades in production-level codes. The required solution prioritizes process and node fault tolerance for SPMD and message-passing models by only inserting a few lines or changing a negligible portion of the code.

1.4 Overview of Presented Research

In this dissertation we aim to address the aforementioned research challenges to reduce resilience costs.

We present Fenix, a fault tolerance framework for SPMD and MPI applications that allows programmers to incorporate on-line recovery into their code. The Fenix framework provides a customizable resilience solution: it readily supports optimized recovery mechanisms if applications want to implement them, but falls back to generic, high performant solutions otherwise. We leverage knowledge from the application to improve scalability, reduce costs, and improve effectiveness.

An overwhelming number of scientific applications can be classified as iterative stencil computations. These applications usually simulate physical phenomena occurring in a domain representative of the reality. They uniformly discretize a continuous space in a mesh or grid of points and statically assign tiles of these points to processors. Typically, stencil applications are composed of a number of iterations, or timesteps, each of which consists, in turn, of two key parts: (1) computation on local tiles to advance the simulation, and (2)

communication with the immediate neighbors. In order to add resilience features, applications that currently use traditional C/R could be optimized using the on-line global recovery method described above. However, due to the locality of the computations in stencil codes, this solution is sub-optimal and its cost can be significantly decreased. In particular, communication patterns in stencil applications imply that, upon failure, by allowing survivor processes to continue the simulation, only the processes directly neighboring failed processes are immediately affected by that failure and are, therefore, the only processes needed for the recovery procedure.

We therefore provide a method that dynamically tolerates process and node failures while only notifying a minimal set of ranks and enabling the vast majority of unaffected computational resources to continue execution without being required to take part in process or data recovery procedures.

Some examples of stencil-based parallel applications that could benefit from such a type of recovery are partial differential equation (PDE) solvers using finite-difference methods, such as applications tracking how temperature spreads in time through a particular part of a 3D space.

If applications with locality features use a local recovery approach to tolerate a failure, the effect of such a failure may propagate slowly throughout the machine. This contrasts with global recovery constructs which force the failure effect to be immediately propagated to all processing elements.

If subsequent failures occur at a distant node before the original failure delay has spread to that node, the delay of the second set of failures will be masked with the delay of the first failure. Therefore, their combined effects on end-to-end execution time will only be the maximum of the recovery delays, instead of their sum. In general, the overhead of multiple, separate, independent failures on the total execution time can appear to be the same as the overhead of a single failure.

This provides a critical advantage for future iterations running on highly unreliable HPC environments: the probability of multiple failures to be masked increases with higher compute resource count and, therefore, there is an opportunity for the recovery delays to be masked and appear as if the application was running on a quasi-reliable machine.

The failure masking effect is studied, described, modeled, and simulated in order for applications with locality features to take advantage of it. Furthermore, we present application algorithms and optimizations that can take advantage of this effect to try and increase performance in faulty environments by leveraging masking advantages.

1.5 Contributions

This dissertation aims to address the challenges mentioned above. In summary, application-aware resilience and fault tolerance methods will likely be required at scale since the cost of providing the application with an abstraction of a failure-free machine, either via hardware or software techniques, will be unsustainable.

The main contribution of the presented work is a set of application-assisted resilience techniques that offer better scalability and lower costs than state of the art techniques. A byproduct contribution is Fenix, a scalable, efficient, flexible, and extensible framework to enable on-line recovery. Applications may leverage their particular communication characteristics to enable optimized failure recovery modes, such as local recovery, within the framework.

Specific research contributions are described in the following subsections and are grouped as follows: global recovery, local recovery, and failure masking.

The expected impact resulting from the presented research is the enabling of domain scientists to focus on the implementation of an application problem itself while minimizing the additional efforts required to optimize the costs of resilience and fault tolerance. Furthermore, these contributions are expected to decrease the overheads of fault tolerance through the use of application-guided resilience as well as to enable an alternate resilient computational model towards exascale.

1.5.1 Global Recovery

This area of research is focused on on-line recovery techniques that make use of all computational resources to collaboratively and transparently (i.e., without the aid of the application) repair an MPI environment. The developed techniques enable transparent on-line process

recovery for parallel SPMD applications experiencing rank or node failures. These applications can then make use of the memory contents belonging to surviving processes at the time of failure detection, as well as previously saved checkpoints, for application-specific data recovery.

Even though these techniques are generic and can be applied to most applications, this dissertation uses iterative finite difference solvers to experimentally demonstrate their advantages.

Specific contributions include:

- **Design of an algorithm for enabling parallel applications to transparently recover from multi-process failures in an on-line manner.**

  This dissertation presents an algorithm that instantiates a specific on-line recovery mode: upon failure, the algorithm captures the failure notification without the need to modify the runtime and the application code, propagates the notification to the rest of the domain, recovers the computing environment by re-spawning new executables or using spare resources, and guarantees that all processes return to a specific, well-known position in the code. At this well-known position, the application may perform particular, application-specific recovery procedures, which can range from the traditional method of reloading a previously-saved checkpoint to approximating the lost data and continuing execution without the need to reload any data.

- **Analysis of application-driven implicitly coordinated checkpointing.**

  This dissertation analyzes how to properly create distributed, consistent checkpoints in an application-aware manner so that the coordination is implicit and communication-less. This approach leverages application semantics to eliminate coordination costs altogether.

- **Formal specification, prototype implementation, integration, and evaluation of the on-line global recovery model with real-world simulations.**

  The implementation discussed in this dissertation provides applications with programming and runtime support for hard failure on-line global recovery as well as an

optional implicitly coordinated, in-memory checkpointing approach to fulfill data recovery requirements. The evaluation of this implementation then demonstrates the framework's ability to support sustained performance in spite of node failures injected at high frequencies.

- **Definition of a resource allocation strategy to optimize throughput for scalable applications with fixed problem sizes while balancing resilience and performance.**

  Some applications require the usage of the entirety of the memory in their allocation. In this case, checkpoints need to be saved in a slower, centralized storage. This dissertation explores the tradeoff between reduced failure rate when using the exact number of nodes necessary to fill the entire memory of all resources and increased checkpoint performance due to in-memory storage enabled by a larger resource allocation to free memory.

- **Study of the tradeoff between resilience and simulation resolution.**

  Some applications with high memory usage can reduce the resolution in certain areas of their domain so that resilience cost can be significantly decreased by enabling the use of more efficient data recovery techniques, such as in-memory checkpointing.

1.5.2 Local Recovery

This body of research focuses on making use of well-defined, local communication behaviors in an application. In such environments, local recovery mechanisms can be implemented to provide optimized and highly scalable on-line recovery. This dissertation demonstrates this concept using an important class of applications: finite difference computations with stencil operators. Special focus is given to isolating and minimizing the number of ranks that are aware of a failure and, therefore, part of the process and data recovery procedure. The goal is for this number to be directly proportional to the failure size instead of dependent on the total domain size. In other words, the number of ranks that take part in the recovery process needs to be constant, $O(1)$, with respect to total domain size.

Specific contributions include:

- **Design and development of efficient mechanisms for leveraging particular communication patterns present in scientific simulations to effectively support failure recovery in a localized manner.** Application-guided local recovery allows unprecedented scalability while the integration with applications can be done using the same programming interface as in the global on-line recovery case.

- **Prototype implementation, integration, and evaluation of the local recovery model** into scientific simulations to allow processes not affected by a failure to be agnostic of it.

1.5.3 Failure Masking

The third set of research activities focuses on studying the different aspects of failure masking as well as increasing its probability of occurring. When several failures occur in certain applications with locality features (for example, stencil computations) using local recovery techniques, there is a certain probability that the total overhead of failure recovery on end-to-end execution time will be less than the addition of each failure's individual overhead. The probability of this effect increases with the machine's component count, which is a highly desirable property in the pursuit of exascale. The presented research activities focus on describing the failure masking effect, experimentally demonstrating its existence, modeling it, simulating it, analyzing its probability of occurring, and studying certain application run time patterns that allow for a further increase of its probability of occurring. The latter study focuses on the exploration of the effects of two different techniques, targeted to stencil codes, that effectively increase the probability of failure masking. Both techniques focus on extending the time it takes for a generic delay (for example, one caused by failure recovery) in a single stencil cell, or group of stencil cells owned by one or multiple ranks, to reach other cells across the domain. The first technique explores how a reduction of the communication frequency between the ranks of an application can dramatically increase the probability of failure masking. The second technique explores how this probability can be further increased by mapping domain cells to ranks in an architectural-aware manner. This dissertation formulates both techniques, analyzes their impacts on failure recovery delay propagation, implements and integrates them with applications, and evaluates both

techniques using real stencil codes at scale while injecting real node and multi-node failures.

Specific contributions include:

- **Description, modeling, analysis, simulation, and experimental demonstration of the existence of the failure masking effect on stencil-based applications integrated with on-line local recovery.**

  When multiple failures are recovered in a local manner, the total overhead on end-to-end execution time might appear as if only a single failure occurred. This dissertation describes the concept, empirically demonstrates it at scale, and provides analytical models as well as simulations and probabilistic analyses of different levels of failure masking.

  Furthermore, this dissertation provides both a quantitative comparison of global and local on-line recovery for an increasing number of failures during an application execution and an analysis of the probability of failure masking as a function of failure density and various application characteristics.

- **Resize of stencil ghost regions to increase failure masking probability.**

  By understanding certain application process communication characteristics and runtime patterns, specific algorithms can be designed to maximize the propagation delay of failure recovery and, therefore, effectively increase the probability of failure masking. Increasing the ghost region size of all ranks artificially decreases the communication frequency. For example, by doubling the ghost point region size, the communication can occur every two iterations instead of every iteration. This dissertation studies and evaluates this technique to demonstrate how the probability of failure masking can be increased.

- **Optimization of rank to node mapping to maximize the effects of failure masking.**

  This dissertation studies how to optimally map the domain regions of simulations to processes so that logically neighboring domains are allocated in processes that are running in the same node. This ensures that node failures have minimal impact in the propagation delay, thereby increasing the effect of failure masking.

1.6 Outline of this dissertation

This dissertation is structured as follows.

Chapter 2 provides the background and a review of related work focused on resilience methods that have been presented to tolerate failures on large scale applications running on HPC environments. It also compares and discusses advantages and disadvantages of the different fault tolerance methods that have been deeply studied in the literature.

Chapter 3 presents a use case study based on production executions of a scientific parallel simulation on top of a high-end HPC environment. This experience serves as a motivation, since it studies the resilience techniques currently in use for production-level applications, the amount of fault tolerance overheads that scientists are willing to tolerate, as well as offers a first-hand experience regarding real failures occurring in top HPC systems.

Chapter 4 describes the research on global recovery performed in the context of the Fenix framework. Fenix provides generic on-line and transparent recovery from process, node, blade, and cabinet failures for MPI-based parallel applications, guided by optional application-specific, implicitly-coordinated checkpoints.

Chapter 5 extends the work on the on-line recovery technique presented in Fenix to present results on local recovery in the context of FenixLR. FenixLR can be used in applications with particular communication characteristics, such as stencil-based parallel applications.

Chapter 6 offers a conceptual and experimental description of the benefits of failure masking, occurring when some application types recover from failures in a local manner. Failure masking allows the overhead of several failures to appear in the end-to-end execution time as if less failures occurred, a highly desirable property toward future extreme scale machines. This chapter also presents models and simulations of the different aspects related to failure masking as well as optimizations that allow its probability to increase.

Chapter 7 concludes the research presented in this dissertation by summarizing the main contributions, their implications and expected impact, as well as outlining directions for future work.

Chapter 2

Background and Related Work

Different failure modes and their characteristics are well documented [139, 148]. A failure instance can be characterized by its domain (the component that has failed, either hardware or software), a certain persistence (it may either halt execution, or it may simply cause erratic behavior), its detectability, and its consistency. A fault is active or dormant depending on whether it causes an error or not; is permanent, transient, or intermittent depending on its presence; and may or may not be systematically reproducible. A large area of research addresses specific kinds of failures, such as silent errors (e.g., silent data corruption or SDC) or process/node failures (e.g., fail-stop errors of one or more processes). Both process and node failures, also known as hard failures, can be caused by different faults. This dissertation focuses on hard failures and considers node failures to be a subset of process failures, since it essentially involves losing a set of processes (i.e., all the processes that are running in a node).

2.1 Hard Failures in HPC Centers

Modern HPC systems, which can compute at petascale-level (i.e., at the order of 10^{15} FLOPS), measure values for the mean time between failures (MTBF) in terms of hours. It has been observed that failures tend to occur in bursts separated by long periods of high stability [146]. An example of this effect can be seen through previous work done on Titan Cray XK7, a system with a reported average MTBF of $\sim$ 7.75 hours [146], where nine failures were observed during an execution lasting 24 hours, averaging $\sim$ 2.67 hours [66]. This discrepancy can be explained through the study of failure distribution. Table 2.1 presents a discrete distribution of the inter-failure arrival period for three HPC systems at ORNL and LANL, including Titan. It is important to highlight that 29% of Titan

Time between two failures (h)	OLCF n_i (%)	OLCF N_i (%)	LANL18 n_i (%)	LANL18 N_i (%)	LANL19 n_i (%)	LANL19 N_i (%)
≤ 1	29	29	18	18	14	14
$1-2$	11	40	13	31	11.5	25.5
$2-3$	7.5	47.5	9.5	40.5	9	34.5
$3-4$	8.5	56	8	48.5	7.5	42
$4-5$	7	63	7	55.5	7.5	49.5
$5-6$	5	68	5	60.5	7	56.5
$6-7$	4	72	4.5	65	5.5	62
$7-8$	3	75	4	69	4.5	66.5
MTBF	∼7.75 h		∼7.5 h		∼7.9 h	

Table 2.1: Frequency of time between consecutive failures for three different systems. Left-most column represents the number of hours between two consecutive failures. Three systems have been studied, all having an MTBF between 7 and 8 hours. For each presented system, n_i represents the relative frequency, as a percentage of all observed failures, and N_i shows the relative cumulative frequency, also as a percentage. Data has been approximated to the nearest half percentage, based on work by Tiwari et al. [146], which are obtained from analyzing proprietary failure logs; the OLCF system refers to the Titan Cray XK7 and is based on six months of failure logs; the LANL systems are based on data collected for nine years.

failures occur within an hour of a previous failure, and around 3/4 of all failures occur more frequently than the MTBF. In the LANL systems, similar results can be seen. An important conclusion is that current systems experience temporal locality of failures: a failure f_2 that follows another failure f_1 will occur with a high probability within one hour of the original failure f_1.

Predictions for the failure frequency of future systems, such as exascale-level HPC resources, vary significantly. For example, some studies predict MTBFs in the order of minutes [49] while others consider future MTBFs to be only slightly lower than petascale observations. This dissertation works on the assumption that there will be some level of increase in both the frequency of failures and the frequency of aforementioned failure bursts, but the conclusions are not based on any specific increase level.

With this increase in MTBFs, the abstraction of a failure-free machine will quickly become infeasible [79]. In order to reduce fault tolerance overhead at future extreme scales, application-aware resilience will become a necessity.

Failure distribution. A Poisson process is typically used to model the distribution of failure arrival times [41, 105, 137, 82, 98, 45, 34]. Furthermore, this dissertation considers the failures to be uniformly distributed among system processes.

2.2 Application-agnostic Techniques

2.2.1 Checkpoint and Restart

Checkpoint and restart (C/R) [85, 84, 82] is the most commonly used technique in order to provide fault tolerance in HPC systems to obtain an application-agnostic solution. In a C/R implementation, the application state is periodically saved (e.g., using BLCR [77] or application-level C/R) so that, upon failure, execution can be resumed from them [15]. In the event of a failure, execution can globally resume from the last saved set of checkpoints by accessing the stored information, instead of restarting from scratch. With this technique, only the work done since the last checkpoint is lost due to the failure; work done up until that point has been "saved". This recovery model has been practiced in HPC for several decades due to its simplicity and adaptivity to the major parallel programming models (such as MPI [65]) which, by default, are designed to abort all active application processes upon a process failure.

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.

Frequency of checkpointing. A large body of literature is devoted to dynamically determining the optimal time between two consecutive checkpoints. A study by Young [154] modeled the checkpointing process and calculated the optimal checkpoint interval that minimizes the waste, assuming the node MTBF is known. Daly [41] improved Young's model by taking into consideration the restart time. In both cases, the authors create a model for the cost of the end-to-end execution, and obtain the optimal interval between checkpoints by minimizing the cost. Later studies focus on experimentally evaluating the results of such models [137], and improving it by including a varying checkpoint interval and modeling the failure arrival time with distributions other than Poisson (e.g., Weibull,

Exponential, Gamma, or Lognormal), under the assumption of reliability awareness [105].

Thanks to the study of the statistical distribution of failures, recommended intervals are based on probabilistic failure models [124] and also take into account dynamic memory usage of applications to reduce network usage. Some approaches [12, 13] minimize the number of checkpoints in respect to traditional iterative checkpoint techniques while others [114] automatically adapt the checkpoint period using dynamic information of the system failure rate.

Consistency of distributed checkpoints. Another property that C/R systems must ensure is **global consistency** of checkpoints. While each compute node checkpoints individually, the different saved states must be coordinated with each other in order to be usable for failure recovery: random, asynchronous memory dumps can often be unusable due to message inconsistency, and, therefore, they can be wasteful. Thus, a communication channel must be used for sychronization. Studies like [82] and [113] distinguish three desired states of a communication channel: in a *consistent state* no internode message sent is out of sequence, in a *transitless state* there are no messages sent that have not yet been received, and a *strongly consistent state* is both consistent and transitless. To ensure that the channel is either consistent or strongly consistent during checkpoint creation, **coordination protocols** must be defined. *Fully coordinated* techniques [54] are used due to their simple implementation. *Non-blocking coordinated* techniques [22, 31], also known as distributed snapshots, build on the assumption that the network channels between all peers are FIFO. Each compute node can begin the checkpoint process only when it has received notification from all other nodes in the system (based on Chandy-Lamport global state research [31]). In *blocking coordinated* techniques [39], the channel is stopped after the reception of the notification until the checkpoint is finished. 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** [21] do not require 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. *Communication induced* checkpointing [2] is a middle point between fully coordinated and uncoordinated techniques, avoiding the domino effect. Uncoordinated checkpointing protocols can leverage message logging to avoid the domino effect for piecewise deterministic applications [54], at the expense of logging all the application messages. For send-deterministic applications, only a subset of all the messages need to be logged [131, 72]. Building on these ideas, some techniques recover only a subset of processes upon failure, while avoiding the domino effect, assuming that the application is send-deterministic [72], or use message logging to reduce the overall waiting time upon failure [129]. Uncoordinated protocols force the processors that survived a failure to wait for the failed process to catch up. Therefore, a system-level optimization [19] suggests using these idle cycles to begin the execution of another application from the system queue, therefore increasing the throughput of the system. Interestingly, a recent study focused on its general applicability [63] claims that the effectiveness of the uncoordinated approach depends on the application communication pattern and the performance of underlying I/O subsystems. This result indicates the necessity of frameworks supporting application-aware local recovery techniques.

Checkpoint Storage and Transfer. In general, checkpoints are saved to a location that survives most system failures, known as the *stable storage* [82]. Typically it is implemented via a resilient centralized storage server; its main advantage is that it is able to withstand a complete system failure, while its main drawback is its low performance –i.e., it presents a bottleneck due to huge data motion from distributed compute nodes to a centralized location. Other research has explored storing the checkpoints in the local memory [125, 156], in both local and peer-memory [155], in non-volatile memory [95], in node-local storage (such as SSD) [120, 6], or at different storage layers [112].

In some scenarios, it is required to reduce both space and network bandwidth consumed to transfer the checkpoints. To that end, studies use techniques such as eliminating duplicate memory pages of different processes before they are saved to storage [116], compressing checkpoints [86], aggregating checkpoints [121], or both [87]. Adaptive incremental checkpointing [88] uses separate cores to create a multi-level checkpoint stack through compression. In other scenarios, it is enough to mask the checkpointing time by

overlapping execution of the application with transmission of checkpoints [115] or by autonomically discovering application's memory access patterns and asynchronously creating checkpoints [118]. In some cases, encodings such as XOR or Reed-Solomon [127] are used, mimicking RAID-like resilience [6].

To prevent the intrinsic data transfer bursts of checkpointing, staging areas (a set of computational resources dedicated to store data in memory) have been used to cache data and scatter the I/O operations in time [142, 103]. Checkpoint staging tries to cache the checkpoint data in the local node [122] or a set of dedicated nodes [128, 120, 126] before sending it to stable storage, while checkpoint staggering [32] limits the number of processes that can concurrently write to stable storage. To further alleviate centralization issues, burst buffers (a layer of distributed solid-state or flash devices) can be used as a cache to the PFS, or, whenever possible, to store a copy of the most recent checkpoint to optimize the restart procedure upon failure [102]. However, few HPC centers offer burst buffers, or restrict their access frequency due to limited durability of the underlying devices [104, 52].

Other techniques maintain a duplicate of the process in local memory while checkpointing [144] or at all times [40]. Non-blocking checkpointing uses staging nodes to create a multi-level checkpointing stack [133]. An optimization [112] proposes a highly scalable in-memory filesystem (with performance similar to that of `memcpy`) to tolerate process failures that, when required, pushes to a multilevel stack to extend failure coverage [125]. Furthermore, the authors offer an interface to access the checkpoint data in the local memory using RDMA, which can be used by third-party tools to store it in the memory of a peer node.

Checkpointing in the Software Stack. C/R has typically been an application agnostic algorithm. However; the application, the compiler, the runtime, and the OS may provide useful knowledge to create a partial checkpoint (excluding temporary buffers) or an incremental checkpoint (including only changes since the last checkpoint). As described in [82], C/R requires a non-trivial **infrastructure** of subsystems in order to succeed. C/R can be embedded in the OS, as BLCR (Berkeley Lab Checkpoint/Restart) does. BLCR is a mechanism that can access process information and memory contents in order to store them in a way that can be later replicated. Higher in the software stack is MPICH-V/cl [22], a global checkpoint runtime based on MPICH and the Chandy-Lamport algorithm. Also at

the runtime level, a study presents an extension of OpenMPI that automatically interacts with BLCR [85]. In a related study, the same authors present an interface to standardize the heterogeneous mechanisms of fully coordinated, stable-storage-based full-process C/R [82].

Other Checkpointing Optimizations. Other aspects of checkpoint performance are being actively studied and improved by the community [116]: C/R has been ported to different architectures [26], optimized for GPGPUs [144, 6], its performance has been modeled [14], and its energy behavior has been evaluated [110].

Combining C/R with other techniques described later in this section has also been explored [98]. A typical method is to group application processes into distinct clusters, apply coordinated checkpointing within clusters and message logging between them [130, 73].

C/R has also been applied to help protect applications from silent error detection [4, 114], proactively and preventively [17] or even on areas out of the scope of HPC, such as cloud computing [45, 117].

Failure overheads towards extreme scale. Since the expected number of failures that a system experiences is proportional to system size, traditional offline 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,

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

Dividing by T, we will obtain the relative overhead, which is

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

The above result shows how, in this scenario, the total overhead is proportional to the size of the system N_S, which is suboptimal. Some of the techniques presented in this dissertation (in particular, those presented in Chapter 5 and Chapter 6) aim to reduce the total failure overhead so much so that it becomes proportional to failure size rather than system size.

2.2.2 Message Logging

Message logging [18, 82] 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* [54]. 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 [82]. In *pessimistic* techniques [21], when a process sends a message, the protocol blocks computation and message delivery until the log is written. *Optimistic* techniques [92] 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 [53] 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 before resuming computation. For example, a study by Lifflander [93] 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.

One problem of message logging is that, upon failure, the failed processes/nodes have to be restarted from the beginning of the simulation, which can cause a huge penalty in all the survivor processes for tightly coupled applications. To minimize this problem, message logging is sometimes used in combination with uncoordinated C/R [22], which allows the application to recover to the state immediately preceding a failure, but incurs a higher failure-free cost. MPICH-V [20] implements two pessimistic message logging protocols and a causal version.

2.2.3 Redundancy

The main idea of redundancy, or replication, [135] is to concurrently (or close to it) run one or more replicas of the data and the computation of a process. When the main process fails, its role can be easily transferred to another replica without the need to roll back or block computation. Levels of replication greater than or equal to 3 are called *N-Modular Redundancy (NMR)* and allow detection, location and correction of silent errors. MPI runtimes that automatically apply redundancy techniques, such as rMPI, MR-MPI, or redMPI, are being designed and implemented. The overheads of various fault tolerant techniques have been studied, and the tradeoff between rollback-recovery and replication has been shown [14]. Other studies [62] also motivate redundancy before C/R, arguing that redundancy can improve MTBF and thereby enable traditional coordinated C/R in exascale systems. Another advantage pointed out in [62] is that redundancy can naturally detect not only fail-stop failures, but silent software and hardware errors (by simply comparing replicas). A later publication [29] claims that the process followed in [62] is incorrect and recalculates the results, concluding that replication is less beneficial than claimed by the original study. The same research group later showed [16] that for a given application, traditional C/R may not scale to exa-scale. For these scenarios, the authors show that replication is not as wasteful.

2.2.4 Process Migration

Pro-active Process Migration uses models to predict which nodes or processes will fail, but must be used in conjunction with other techniques to guarantee reliability [151]. It also requires spare nodes as migration targets to continue the work.

2.2.5 Application-agnostic Runtimes

Several fault tolerant MPI runtimes exist, such as MPICH-V [20], rMPI [60], MR-MPI [55], redMPI [64] or FEMPI [141]. All of them implement completely transparent failure recovery, i.e., they offer a vision of a fault-free system to the application. To do so, each one implements one or more of the aforementioned application-agnostic fault tolerance techniques.

However, as explained in Chapter 1 and suggested by recent studies [79], by leveraging the knowledge from the application, constructs that are more scalable and efficient can be designed.

2.3 Application-aware Techniques

Algorithm-Based Fault Tolerance (ABFT) was first proposed by Huang and Abraham [81] to detect, locate, and ideally correct silent errors in matrix operations. These errors do not halt computation, but do produce incorrect results.

ABFT consists of augmenting matrix data with an extra checksum column and/or row. Several matrix operations, such as matrix-matrix multiplication [74], have been proven to possess the property that the checksum relationship from the input is maintained in the output. Thanks to this property, a failure can be detected after the operation, by checking if the checksum is consistent. Several matrix-matrix multiplication algorithms (Cannon, Fox, and outer product) are studied in [35], and implemented and evaluated on GPGPUs in [47]. QR factorization is also shown and evaluated for GPGPUs [51]. Other operations, such as Hessenberg Reduction (HR) [89], right-looking LU factorization [44], Cholesky factorization [76], Krylov subspace iterative methods [34], FIR (Finite Impulse Response) filtering [81], Fast Fourier transform [152], sorting [37], and median filtering [150] have been proven to work with ABFT techniques. More recently, studies like [35] present an extension of this scope to also support fail-stop failures. The protocol maintains a checksum of the significant data of all the processes and the application operates in such a way that the checksum relationship continues to hold at all times during the computation. If the data on a node is lost, the data is recovered without the need of a checkpoint by simply unfolding the checksum operation. In contrast with integer operations, however, during floating point calculations – where computation is inexact – inconsistent checksums due to rounding (and not due to silent errors) have been detected; it is difficult to distinguish between these. In some cases, silent errors can be detected, located, and corrected on-line [47, 44], rather than correcting them at the end of the calculation (off-line).

If an application can converge and achieve a correct (or approximately correct) outcome even though some information is lost during the process, it is **naturally fault tolerant**.

This is a highly desirable property of a fault tolerant technique. However, not all the algorithms can be fault tolerant by nature.

In other application categories, such as bag of tasks (or master/worker architectures), fault tolerance can be implemented by assigning a failed task to a new resource [101], without the need of checkpointing.

Containment Domains (CD) [38] involve augmenting applications by wrapping every function or subroutine with preserve, detect, and recover recipes, to prevent failures from propagating out of the subroutine from which they originate. CD is a language extension which enables the specification of failure detection, data preservation and recovery schemes for different code blocks in a nested and hierarchical manner. While CD present a generic and abstract construct able to fit many kinds of applications and failure types, in this dissertation we focus on hiding the details of state preservation, failure detection, and failure recovery by restricting to a specific kind of applications (i.e., those whose state can be specified by a set of data chunks and that contain strongly consistent states) and failures (i.e., process/node failures).

Resilience in the MPI standard. To enable algorithms to work for fail-stop failures, the MPI standard should offer a set of fault tolerance mechanisms. Dynamic features of MPI 2, such as process re-spawning or communicator merging, would help the recovery process, but MPI 2 lacks basic tools such as failure detection and survival. Even newer versions of the standard, e.g., MPI 3, do not include such features yet.

Two main extensions have been suggested to support process failures. HARNESS FT-MPI [58] was an experimental complex implementation (discontinued in 2003) that supported $n-1$ node failures on an n-node execution through several modes of operation. Based on one of the recovery modes of FT-MPI, **User Level Failure Mitigation** (ULFM) [11, 10, 83] is being proposed as a minimal set of changes to the MPI Forum.

Classifying Failure Recovery Techniques. We classify recovery mechanisms in four broad categories, disjoint two by two.

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 all the approaches presented in this dissertation implement on-line recovery

constructs.

When using **roll-back** protocols, recovering from a failure triggers all processes (including the spare processes) to revert to a valid state in the past, at which point the computation is restarted from that state. Traditional C/R is a roll-back recovery mechanism. When implementing **roll-forward** protocols, however, survivor processes are allowed to reconstruct a valid state without the need to globally revert the state to a previous consistent state. In particular, when a failure is detected, all processes construct a valid state, or potentially valid after some extra computations. Algorithm-based Fault Tolerance techniques [150, 152, 17, 47, 50] fall into this category.

The software framework presented in Chapter 4 show how efficient execution in failure-prone scenarios can be enabled by using global online recovery in conjunction with advanced in-memory diskless checkpointing. To fill in failed processes in an online manner, this framework leverage a spare process pool which is readily integrated with the in-memory checkpointing mechanism. Despite its usefulness demonstrated with large scale stencil computations, global rollback and synchronization are still necessary even for the response to a single process failure. Chapter 5 and Chapter 6 presents a design of a scalable recovery mechanism for stencil computations that allows these issues to be overcome.

Chapter 3

Understanding Node Failures on Extreme-Scale Production Runs

This chapter presents the study of a production execution of the S3D combustion simulation on 130 thousand cores during a span of 24 hours on Titan. The goal of this chapter is to analyze the behavior of current fault tolerance techniques upon the event of real failures, and to highlight the failures encountered and the fault tolerance actions taken.

S3D [33] 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 [25]. It employs realistic gas-phase thermodynamic, molecular transport properties and detailed chemical kinetics by interfacing with the CHEMKIN library [97]. 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.

3.1 Extreme Scale S3D Production Execution

This section analyzes failures and fault tolerance methods used in a production execution of the S3D application on Titan.

Figure 3.1 shows all events related to fault tolerance and failures that occurred during a 24-hour production run of S3D. This particular execution simulated a 3D domain that was decomposed and distributed following a $50 \times 52 \times 50$ pattern. It was executed using 130,000 cores on the Titan Cray XK7 system at ORNL, which represents 8,125 compute

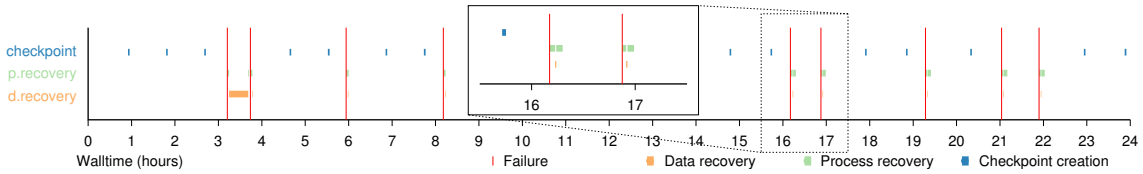


Figure 3.1: Checkpoint and failure timestamps on production runs using 130,000 cores on Titan. The x-axis represents the walltime in hours. Each row, from top to bottom, indicates a particular event related to fault tolerance: the first row indicates when data is being checkpointed, the second row indicates that the MPI processes are being recovered, and the last row indicates that data is being recovered. Vertical red lines indicate the times in which a failure occurred ($\pm$5 seconds)

nodes, or $\sim$44% of Titan's compute nodes. Titan is composed of 18688 16-core CPUs 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.

The execution experienced nine node failures distributed throughout the 24 hours, which were recovered by stopping all MPI processes, restarting the MPI application, and reloading checkpoints that were previously saved in the parallel file system. Each checkpoint saved 5.2 MB/core of important data –saving an aggregate of 660 GB/checkpoint. The checkpoint operation took 55 seconds while the checkpoint loading operation took 44 seconds, and used IO technology as described by the ADIOS project [103, 90]. This shows an empirical PFS write bandwidth of 12 GB/s and a read bandwidth of 15 GB/s. Table 3.1 summarizes all the overheads related to failures and fault tolerance, categorizing them depending on its source.

Note that the overhead of checkpointing was 1.72% compared to a failure-free, checkpoint-free execution, which was estimated by subtracting overheads from the total time of 24 hours. The rollback overhead was 22.63%. Together, all effects due to fault tolerance and failures accounted for 31.40% when compared to the useful computation.

The checkpoint frequency for the described production runs was set to one checkpoint every 600 iterations (each iteration lasting around 5.30 seconds) because the MTBF was estimated to be higher than what it actually was. This can be explained by the fact that

	Total	Overhead/Useful
Useful iterations	65,752 s	
Checkpoint	1,126 s	1.72 %
Process recovery	4,242 s	6.45 %
Checkpoint load	396 s	0.60 %
Rollback	14,884 s	22.63 %
Total overhead	20,648 s	31.40 %

Table 3.1: Overhead and useful computation times of the S3D production run depicted in Figure 3.1. The right-most column indicates the percentage overhead when compared to a failure-free and checkpoint-free execution.

node failures in HPC environments strike in frequent bursts, and are separated by longer periods of stability [146], as described in Chapter 2. This can make it hard to predict the checkpoint frequency that will provide the optimal benefit in all cases. In particular, the presented execution experienced nine node failures and, therefore, a higher overhead than expected. As mentioned, the total overhead due to checkpointig is only 1.72%, while the rollback overhead is of 22.63%. Since there is a clear tradeoff between these two metrics, it is understood that if checkpoints are performed more frequently (i.e. more checkpointing overhead), the rollback cost will decrease.

This chapter presents a simulation to better understand what would have been the best checkpoint rate for the particular case of those nine failures. Such a simulation has been performed (1) first assuming each failure was going to strike at the exact same time as observed in the production runs and (2) later injecting a significant number of combinations of nine failures within the 24-hour period that the execution expanded to obtain a more realistic estimate.

3.2 Modeling Production Run Behavior

A number of studies, such as those presented by Young [154] or Daly [41] have already explored what is the optimal interval between consecutive checkpoints depending on a number of application and system characteristics –such as checkpoint time, expected failure rate, and, in some cases, recovery time. Our approach, however, requires to first understand what

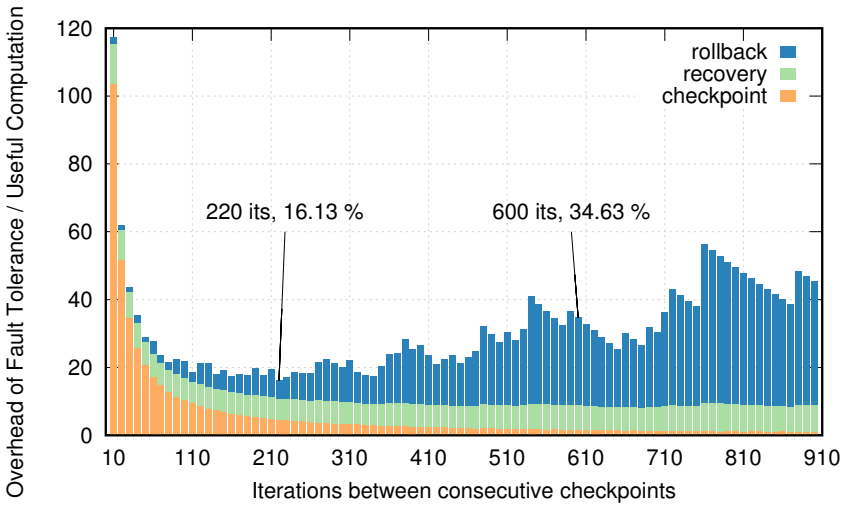


Figure 3.2: Simulated overhead results of injecting nine failures at the exact same timestamps as occurred in the production runs (see Figure 3.1) while changing the checkpoint frequency (x-axis). y-axis shows the stacked overhead compared to computation that was useful. Checkpointing every 220 iterations would have been optimal in the case of these particular nine failures.

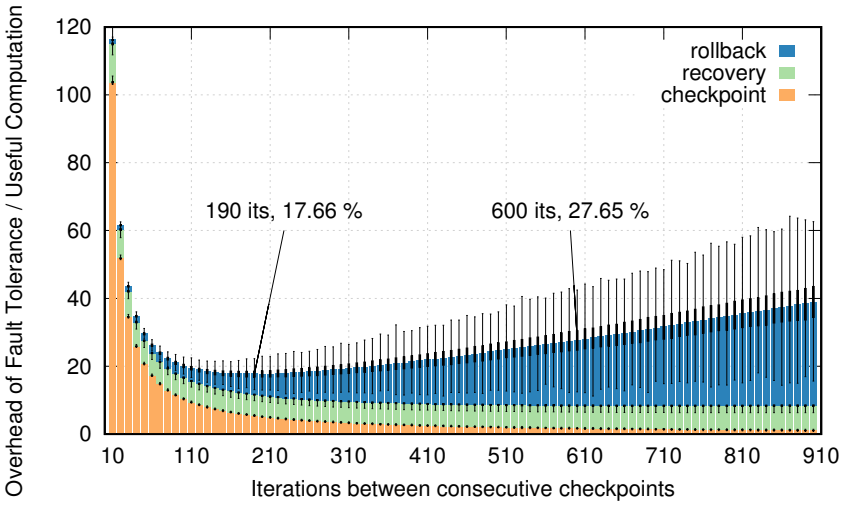
would have been the total overhead for different checkpoint frequencies, assuming that the node failures occurred at the exact same timestamp as they did on the production runs. This is shown in Figure 3.2.

Increasing checkpoint frequency. As seen in Figure 3.1 and Table 3.1, data was not checkpointed frequently enough to efficiently tolerate the high number of failures that occurred. Figure 3.2 shows the overhead of fault tolerance compared to the total failure-free and checkpoint-free execution for different checkpoint frequencies. The overhead has been categorized in (1) rollback overhead –i.e., the time lost between a failure and the last checkpoint preceding that particular failure–, (2) recovery overhead –i.e., the time to detect the failure, turn off all MPI process and the MPI runtime, restart all MPI processes, and reload the checkpoints from the PFS–, and (3) checkpoint overhead –i.e., the time to save the important application state to the PFS. To estimate the recovery overhead, we added the average recovery overhead observed in the production runs (i.e. 471 s) and the time to load the checkpoint from the PFS and distribute it among all MPI processes (i.e. 44 s).

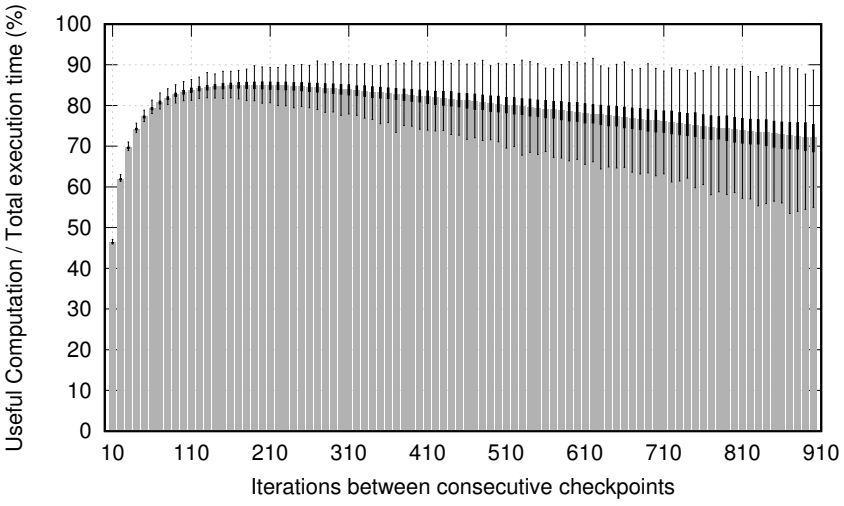
The conclusion of this experiment is that an optimal checkpoint period of 220 iterations between consecutive checkpoints would have offered a minimal total overhead of 16.13%, when compared to a failure-free and checkpoint-free execution. Note that without previous

knowledge of the particular failure count that will impact the system it is less costly to choose a checkpoint frequency that is lower than one that is too high. However, note that the simulation estimated an overhead of 34.63% when checkpointing every 600 iterations, while Figure 3.1 shows an overhead of 31.40% for these particular nine failures.

Modeling failure injection times. While Figure 3.2 offers an important result showing that the optimal checkpoint period is of 220 iterations, it does so by assuming a particular combination of nine failures. The next step in this study focuses on understanding the effect of any combination of nine failures that may occur in a 24-hour period. Assuming that a failure can only occur exactly every 60 seconds (i.e. every minute), simulating all possible combinations would require simulating $\binom{24\times60}{9} = 7.15 \cdot 10^{22}$ combinations, which is not practical. Instead, we performed ten thousand repetitions of the experiment, assuming that each failure is uniformly distributed among the 24 hours. To do that we used a random number generator from the C++ standard (in particular, the 32-bit `std::mt19937` implementation of the mersenne twister algorithm by Matsumoto and Nishimura) and the `std::uniform_int_distribution` random number distribution. The results of applying this technique by using the same behavior as in the 24-hour production runs (i.e. total time, iteration time, checkpoint time, and recovery time) can be seen in Figure 3.3a and Figure 3.3b. The former shows the overhead of the nine failures compared to a failure-free and checkpoint free execution. The latter plots the throughput of the simulation, i.e. number of iterations actually finished and not lost due to rollback caused by a failure. In both cases, the height of each bar indicates the average of ten thousand repetitions while the error bars in both figures show the variability between those repetitions: minimum, maximum, first quartile, and third quartile. As can be seen in Figure 3.3a, the variability of the total checkpointing cost among the repetitions is minimal, which is expected. The total recovery overhead is similar in most cases, but, in some infrequent scenarios, a failure can occur while a previous failure is still recovering, effectively truncating the recovery time of the first failure as the new recovery process starts (this can be seen by observing the minimum value in each recovery overhead observation). Finally, the greatest source of variability is due to the rollback overhead. Optimistically, all failures will occur right after a checkpoint is finished, with a total rollback overhead of zero. Pessimistically, all failures



(a) Overhead, categorized depending on its source.



(b) Throughput.

Figure 3.3: Simulated results of randomly injecting nine failures while changing the checkpoint frequency (x-axis). Error bars show variability of performing 10,000 repetitions. Checkpointing every 190 iterations offers an optimal throughput (and minimal overhead) for the average

will occur right before a checkpoint is finished and, therefore, unusable. In this case, the total rollback overhead is almost $9 * (T_{it} \times N_{it,C} + T_C)$, where T_{it} is the time to compute an iteration, T_C is the time to checkpoint, and $N_{it,C}$ is the number of iterations between checkpoints. Henceforth, the average case is when all failures occur exactly halfway between two consecutive checkpoints, incurring a total rollback overhead of $9 * (T_{it} \times N_{it,C} + T_C)/2$. As can be observed in the error bars of Figure 3.3a, 50% of the repetitions incur a rollback overhead close to this average.

The minimum overhead of 17.66% when compared to a failure-free and checkpoint-free execution can be achieved by checkpointing every 190 iterations. In this case the average checkpointing overhead of 5.42%, when compared to the useful computation, is similar to the average rollback overhead of 6.09%. This contrasts with the same overheads in the real production runs, which were of 1.72% and 22.63%, respectively, as discussed in Section 3.1. The average simulated recovery overhead is 6.14%, while in the production runs the process recovery overhead was observed to be 6.45% and the data recovery overhead was 0.60%, adding a total recovery overhead of 7.05%. The difference is, as noted before, due to the fact that in some repetitions, one or more failures can happen while a previous failure is still not recovered. These repetitions pull the average recovery time down from the parametrized values. This result coincides with the maximum throughput of 84.98% (see Figure 3.3b), which can also be obtained when checkpointing every 190 iterations. Note that this simulation estimated the average overhead of nine random failures with the same parameters as in the production runs (i.e. checkpointing every 600 iterations) to be 27.65%, which is similar to the actual overhead for the particular nine failures that actually occurred (31.40%). The difference can be ignored, as the observed overhead for the nine failures was one particular instance, while the simulated 27% overhead is the result from averaging 10,000 repetitions.

This study helped understand the behavior of S3D production runs, which is critical when evaluating the presented techniques and determining sensible parameters for the models developed in this dissertation.

Chapter 4

Application-aware On-line Global Recovery

The goal of this chapter is to design, implement and evaluate a framework for enabling recovery from process/node/blade/cabinet failures for MPI-based parallel applications in an on-line (i.e., without disrupting the job) and transparent manner. The developed framework, called Fenix, provides mechanisms for transparently capturing failures, re-spawning new processes, fixing failed communicators, restoring application state, and returning execution control back to the application. To enable automatic data recovery, this section relies on application-driven, diskless, implicitly-coordinated checkpointing.

Using the S3D combustion simulation presented in Chapter 3, running on the Titan Cray-XK7 production system at ORNL, this chapter experimentally demonstrates Fenix's ability to tolerate high failure rates with low overhead while sustaining performance.

4.1 Overview

Application resilience is a key challenge that must be addressed in order to realize the exascale vision. Process/node failures, an important class of failures, are typically handled today by terminating the job and restarting it from the last stored checkpoint, as shown in Chapter 3. Some estimations suggest this approach might not scale to exascale.

Framework for Global Recovery. This chapter presents the design, prototype implementation, and evaluation of Fenix[1], a framework aimed at enabling **on-line** and **transparent** recovery from process, node, blade, and cabinet failures for parallel applications in an efficient and scalable manner. Fenix provides mechanisms to transparently capture

[1]Fenix is the old English noun for Phoenix, the mythological bird that, like an application that uses the Fenix library, is reborn from its ashes. It is also the Catalan noun for Phoenix.

failures, re-spawn new processes (or use spare one), fix failed communicators, restore application state, and return the execution control back to the application. As MPI is envisioned to continue being at least one of the de-facto communication libraries on exascale systems [7, 27], Fenix targets MPI-based applications. Fenix can leverage existing checkpointing solutions to enable automatic data recovery. It does, however, provide an in-house implementation of application-driven, diskless, implicitly-coordinated checkpointing. This dissertation uses this implementation to explore how application-driven checkpointing can eliminate the cost of coordination (both explicit or implicit synchronization) under certain assumptions, and can reduce the size of the checkpoints by saving only the essential data.

Fenix has been prototyped and has been deployed on the Titan Cray XK7 production system at ORNL, the world's third fastest machine as of November 2016. The presented prototype implementation of Fenix leverages User Level Failure Mitigation (ULFM) [10, 11, 86, 9, 61], which has been proposed as a minimalist and lightweight MPI extension that allows applications to create policies for tolerating process failures.

Experimental evaluation. This chapter also presents an experimental evaluation of the effectiveness and scalability of Fenix using four benchmarks as well as the S3D [33] combustion application, previously introduced in Chapter 3 on Titan. Results demonstrate Fenix's ability to tolerate high-frequency dynamically injected multi-node failures while maintaining sustained performance of the S3D application. The evaluation also explores extreme execution scenarios that may exist at some highly unreliable time periods of future machines, where node failures occur with high frequency (i.e., as often as every 47 seconds). For example, when injecting node failures every 90 seconds and checkpointing every 3 application iterations (∼5 seconds), performance is sustained with 15% overhead when compared with a failure-free and checkpoint-free execution. Experiments also demonstrate that coordination-less checkpointing scales perfectly up to 250 thousand cores in a failure-free scenario, resulting in a sustained checkpoint bandwidth of ∼17 TB/s when checkpointing every 18 seconds, with an overhead of 0.41% with respect to a checkpoint-free execution. In this experiment S3D simulated 31+ billion grid points, resulting in 2+ TB per checkpoint. This chapter shows how the programming overhead of using Fenix is low, requiring less than 35 new, changed, or rearranged lines of code in S3D. Finally, this chapter

revisits in-memory checkpointing for memory-hungry applications. Discussions with application developers and fault tolerance experts suggest that double in-memory checkpointing may not be desirable. The main criticism is that some applications must use all available node memory. This chapter investigates this issue by exploring costs and benefits of job allocation increase to free part of the nodes' main memory, and presents models and simulations showing how using on-line recovery and in-memory checkpointing may be benefitial even for memory-hungry applications. We compare double in-memory checkpointing with PFS-based checkpointing, which can tolerate applications that require more memory. We also consider an alternate *mixed* approach that uses all memory available to store a part of the checkpoint in a double in-memory fashion, and the rest in PFS. The mixed approach is an idealization that assumes that checkpoint data can be arbitrarily divided between PFS and in-memory storage. However, as will be shown, the point that maximizes throughput is invariably found when the mixed approach stores the entirety of the checkpoint in memory.

Outline. The rest of the chapter is organized as follows. Section 4.2 describes architectural design of the Fenix library recovery mechanisms. Section 4.3 presents its programming interface and integration within S3D. Section 4.4 presents an experimental evaluation of S3D with Fenix on Titan. The chapter ends with Section 4.5 by studying how on-line recovery may benefit memory-filling applications.

This chapter contains portions adapted from published papers by Gamell et al. [66] (adapted with permission) ©IEEE 2014.

4.2 The Fenix Architecture for On-line Failure Recovery

In current implementations of MPI on production systems, node or process failures cause the entire MPI job to fail. This forces applications to recover from such failures in a bulk synchronous manner even if the failure occurs in only one node. S3D recovers from failures by restarting from the last checkpoint, which involves reading the large checkpoint file from persistent storage. This incurs a significant IO overhead, and additional costs for repeating the computations since the last checkpoint. These costs can be non-trivial even at petascale, but will certainly become prohibitive if MTBF reduces at exascale, since the time of checkpointing to disk is typically in the order of minutes for S3D. Emerging

implementations of the MPI standard are expected to include mechanisms to tolerate these faults without incurring a job failure. Leveraging such MPI implementations, specifically ULFM [11, 10], this section presents the design of Fenix with the goal of transparently recovering from process failures in an on-line manner. To enable automatic data recovery, Fenix relies on a checkpoint mechanism that is able to tolerate full-node failures. While applications can use any existing solution to realize data recovery, Fenix includes an in-house implementation of application-driven, diskless, implicitly-coordinated checkpointing. This implementation is presented later in this section.

4.2.1 Process Recovery in Fenix

Process recovery in Fenix involves four key stages: (i) detecting the failure, (ii) recovering the environment, (iii) recovering the data, and (iv) restarting the execution.

Failure detection is delegated to ULFM-enabled MPI. This standard guarantees that all MPI communications should return an `ERR_PROC_FAILED` error code if the runtime detects a process failure in the communicator. The error codes are detected in Fenix using MPI's profiling interface. As a result, no changes in MPI runtime itself are required, which will allow portability of Fenix if interfaces such as ULFM become part of the MPI standard.

The first step in **environment recovery** is to invalidate all the communicators and to scatter the failure notification to all the ranks, as shown in Figure 4.1. In the current Fenix prototype, this is done using the ULFM revoke operator for all communicators in the system – the user must register their own communicators as discussed in Section 4.3.1. After that, the world communicator has to be shrunk to remove failed processes, while the rest are freed. If this step succeeds, new processes are spawned and merged with the old communicator using the dynamic features of MPI 2. As this may reassign rank numbers, Fenix uses the split operation to set them to their previous value. Note that this procedure allows $N - 1$ simultaneous process failures, N being the number of processes running. An alternative approach is to use processes from a previously prepared process pool instead of spawning new processes.

Once Fenix's communicators are recovered, a long jump is used to return execution to `Fenix_Init()`, except in the case of newly spawned processes – or processes in the process

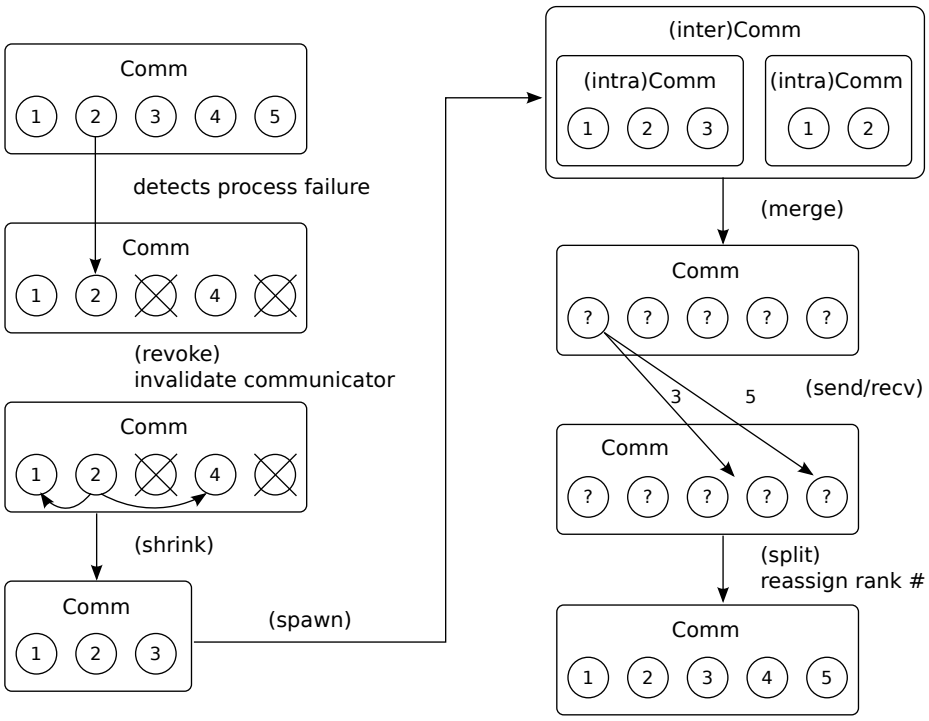


Figure 4.1: Communicator recovery in Fenix by spawning new processes. The recovery process when using a process pool is similar. ©2014 IEEE (reprinted with permission) Gamell et al. [66].

pool – which are already inside `Fenix_Init()`. From there, all processes, both survivors and newly spawned, are merged in the same execution path to get ready to recover the data.

Data recovery in Fenix is achieved using application-driven, implicitly coordinated, neighbor-based diskless checkpoints created as described in Section 4.2.2. The first step towards a safe recovery is to calculate the latest set of checkpoints that are present on all the cores (i.e., that form a strongly consistent global checkpoint) for every saved data element. As Fenix collectively assigns an increasing natural identifier I_C to every new checkpoint (note that this is done in a distributed manner, i.e. without any synchronization/communication), this is done by finding the maximum I_C that is common to all ranks. This number, $I_{C,max}$, will correspond to the last checkpoint successfully completed by all ranks. To obtain the identifiers of the rest of the saved elements, no communication is required; it is only needed to find, for each saved element, the largest checkpoint identifier that is smaller than $I_{C,max}$. Finally, as Fenix uses neighbor-based checkpointing (see Section 4.2.2) to store checkpoints

in a neighbor node's memory, the next step is to copy these checkpoints back to the new process. Once the checkpoints have been copied to the local memory of the newly created processes, the only remaining step is to replace the contents of the original element's memory with the contents of the checkpoint. This is done using `memcpy()`.

Resuming execution can occur once the communicators and the data have been recovered. Execution can return: (i) to the beginning (after `Fenix_Init`), or (ii) to the last checkpoint that completed successfully. Dynamic labeling features are used in Fenix in order to allow static transfer of execution control at checkpoints. The former technique has several design advantages for applications that require complex communicator layout. For example, leveraging initialization code that is already existing in applications for creating communicators, communicators can be recreated after process recovery without the need of writing new code (see Section 4.3.2 for an example). The re-generation code is executed after Fenix returns the execution environment to the predefined point at the beginning of the application. In the prototype integration with S3D the code resumes at the beginning of the execution where when a failure has been detected and recovered. As shown in Section 4.3.2, some of the modules are re-initialized and lost communicators are transparently re-created before going back to the computation in the main loop.

Usage in already fault-tolerant applications. An application may want to use all Fenix tools but, in some cases, only a subset may be enough. For example, ABFT applications that work with augmented data sets usually tolerate only silent errors. The augmented data set, perhaps with some minor modifications, should be enough to recover from a fail-stop failure. In similar situations, the automatic failure detection and transparent process recovery in Fenix can speed up the recovery process.

4.2.2 Application-driven Data Checkpointing

To enable automatic data recovery, Fenix relies on a checkpoint-centric mechanism that is able to tolerate multi-process failures. While applications can use any of the existing checkpointing platforms or any other data recovery methods in the community, Fenix offers an implementation featuring application-driven checkpointing, which enables implicitly coordinated, application-driven, selective checkpoints. Fenix also guarantees strongly consistent

global checkpoints that can be safely used for restarts. Furthermore, Fenix uses neighbor-based diskless checkpointing, which can achieve high efficiency while scaling to a large core count. In addition, this can be achieved without the need for coordination-induced communication either by the checkpoint process or assuming any synchronization done by the application.

Selective Checkpointing. Conventional, application-agnostic checkpointing typically saves the entire state of the application –including all its memory pages, the heap, and the stack– even if they contain more data than necessary for recovery. In contrast, application specific checkpoints do not have to store the entire state of each process; in Fenix, the applications can specify which data elements should be checkpointed. The program counter of each call to the checkpoint operation is also stored along with a copy of the checkpoint to allow the return of the control to that point. This approach allows every element to be independently checkpointed when the application specifies, not as a subset of a larger checkpoint. Ideally, this should be done when the data is updated. The fact that the application checkpoints only part of its data at a time allows to reduce the difference between MTBF and the checkpoint time. As Chapter 3 shows, this is critical towards exascale resilience.

Implicitly coordinated checkpointing. This technique allows the creation of checkpoints locally (i.e., without requiring any communication) while the application is in a strongly consistent state. Strongly consistent states are states in which there are no messages sent after the state that have been received before it and there are no messages sent before the state that have been received after it [82]. In other words, strongly consistent states are states in which no messages cross the point where the state is consistent. Therefore, checkpointing the data locally while in an strongly consistent state guarantees a set of checkpoints from which a proper recovery can be achieved [82]. Application knowledge can be used to identify such strongly consistent states and to create strongly consistent checkpoints without the need for any explicit or implicit communication and without assuming any synchronization (e.g., barrier). This is in contrast to creating checkpoints in an application-agnostic manner where coordination protocols among the distributed application processes

are typically required, which can be expensive and lead to scalability challenges. Uncoordinated checkpointing typically requires support mechanisms such as message logging to avoid the so-called "domino effect", which can occur in non-send-deterministic applications as described in [72]. Assuming that the application calls `Fenix_Checkpoint` (a collective operation that will perform a local checkpoint) while in a strongly consistent state, the set of local checkpoints will form a strongly consistent global checkpoint. In case of a failure, this checkpoint can be used for recovery as if it was created using a coordination protocol. The main advantage of this technique is that it supports highly imbalanced applications without requiring synchronization or any coordination-induced communication, which can degrade performance. Other advantages of this approach include its ease of implementation and efficiency. Fenix simply assigns an increasing natural identifier (cost $O(1)$) to every local checkpoint, and, upon recovery, it calculates the maximum common identifier shared by all processes for every saved element. Note that the cost of checkpoint creation is $O(1)$ regarding the number of nodes, as it does not require any synchronization or communication with any other peer process. This is, therefore, an extremely scalable "protocol" that delivers the same benefits as coordinated checkpointing with the low complexity of uncoordinated checkpointing. The cost of checkpoint recovery is $M \times cost(AllReduce(integer))$, where M is the number of saved elements. This translates to a linear time $O(M \times N)$ depending on the number N of nodes in the worst case of a linear implementation of the `MPI_Allreduce`'s max() function.

In many scientific applications there are naturally occurring patterns that can be used to identify an appropriate location for the Fenix checkpoint operator. For example, for iterative simulation codes such as S3D that contain an outer timestep loop, positioning the checkpoint operator at the end of a timestep would typically result in an implicitly coordinated checkpoint (as shown in Section 4.3). For other application structures, strongly consistent states must be identified to determine where to position the checkpoint operator. Possible locations may be synchronization points, sub-module finalizations, before/after a communication-intensive period, after or during a computation-intensive region, after updating key data, etc.

While the approach does require that the checkpointing operators are correctly placed

within application code, this overhead can be significantly reduced by providing an appropriate high-level programming interface, which will be presented in Section 4.3.

The great advantages offered by these techniques are enabled by keeping the application in the loop. However, as the typical HPC user is a scientist programmer focused on its problem domain, and not an expert in fault tolerance, these tasks may become overwhelming. In Section 4.3 these ideas will be exposed to the user in a simple manner to provide the scientist an abstraction to make the usage of **implicitly coordinated** available and simple. In other words, it will be shown how these advantages can be added in the application with little intrusiveness.

Diskless Checkpointing. As many current high-end systems do not have local storage on the compute nodes (e.g., the top 5 machines as of November 2016 do not have compute node local storage [147], such as a hard drive or an SSD) stable storage is typically provided through a centralized/semi-decentralized parallel filesystem (e.g., Lustre [23]). As diskless checkpointing typically offers faster checkpoint performance [155, 112], and S3D does not require all the memory in the compute nodes, neighbor-based checkpointing [155] has been implemented in Fenix, in which, for every checkpoint, each node stores a copy of its local data plus a copy of the data from a peer node. Fenix considers a group of s_{group} physically-close ranks spanning across a subset of the machine. In order to enable data to survive node, blade, or cabinet failures (or failures of any other subset of the machine), Fenix saves the checkpoints of all ranks in the group in the memory of ranks outside of the same group. For example, with a peer-node size of 96 physical nodes (one Cray XK7 cabinet), the checkpoints of one cabinet will be stored in a neighbor cabinet and as a result, from a data perspective, correlated failures within a cabinet can be tolerated. Assuming s_{total} is the total number of ranks used in the execution, to determine which remote MPI rank r_r will store a node r_l checkpoint, Fenix uses $r_r = (r_l + s_{group}) \bmod s_{total}$, which assumes that the application uses at least 2 entire peer-nodes, but can work on partial peer-nodes beyond that (e.g., 2.5 cabinets). The cost incurred from setting a larger peer-node size in torus-like networks (Gemini [1]) is that checkpoint data will have to traverse more nodes to reach its peer-node.

Currently, only one group failure is tolerated in the worst case scenario, as checkpoints are saved in the ranks of a remote peer-node. In practice, however, assuming a non-correlated distribution of failures among the group of ranks (which is certainly not an unrealistic assumption, as the group size can be tuned to contain correlated failures), if two groups have to fail, the probability of the failure of a node and its peer-node is:

$$\frac{N/2}{\binom{N}{2}} = \frac{N}{2 \times \frac{N!}{2!(N-2)!}} = \frac{(N-2)!}{(N-1)!} = \frac{1}{N-1} \tag{4.1}$$

which for Titan translates to 1/18687 assuming a group of 16 cores (a physical node) or 1/194 assuming a group of cabinet-size (96 physical nodes). Therefore, it is highly unlikely that if two failures occur simultaneously, no recovery by the neighbor algorithm is possible.

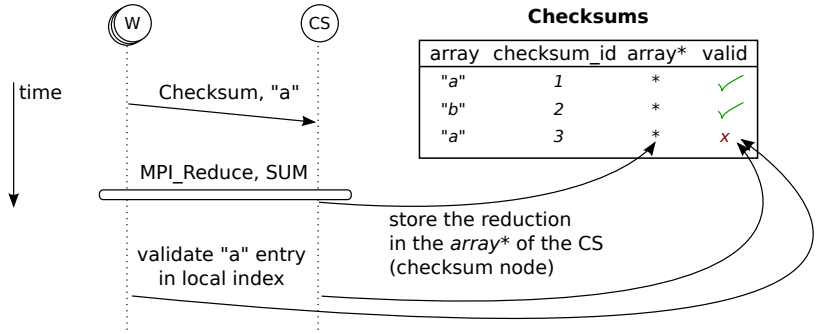


Figure 4.2: An illustration of the checksum-based checkpointing approach.

Furthermore, Fenix also implements checksum-based checkpointing, in which each node stores a local copy of its own data and a dedicated rank stores the bitwise checksum of the data from all the nodes. This process have lower memory footprint in the cores, compared with neighbor-based checkpoints. This process is outlined in Figure 4.2. Note that helper nodes tolerate failures as much as compute nodes do. This differs from stable storage which is assumed to be fault tolerant on its own and, therefore, requires a separate level of data replication.

However, without the use of more advanced encoding codes such as Reed-Solomon [127], this method only supports one simultaneous process failure. Therefore, all experiments are deployed with neighbor-based method.

4.3 Fenix Programming Interface

The API provided by Fenix prototype, for both C and Fortran, is comprised of five operators as described in Section 4.3.1. Section 4.3.2 describes the integration of S3D with the first Fenix prototype, and the changes required to the S3D code to tolerate process, node, blade, and cabinet failures using Fenix.

The current version of the Fenix API (based on version 1.0.1 [70]), can be found in Appendix A.

4.3.1 Interface Overview

The following functions make up the API of the Fenix prototype. Their usage is discussed below.

`Fenix_Checkpoint_Allocate` notifies Fenix about a data element (e.g., an array) that will be saved. Only two parameters are needed, specifying the memory location of the element to save and its size in bytes. It returns an identifier that can be used to uniquely refer to the element when actually performing a checkpoint.

`Fenix_Init` is the basic operation to initialize the Fenix library. Three parameters are required to specify the checkpointing algorithm, its parameters, and the resuming destination. The function returns the `status` of the rank, specifying whether this is the first time the application is started (i.e., `Fenix_st_new`), it has survived a failure (i.e., `Fenix_st_survivor`), or has been respawned after a failure (i.e., `Fenix_st_respawned`; note that processes are considered respawned even if the spare process pool method is used). It also returns `world`, the new/repaired world communicator, which will include all processes except ones kept in a process pool to tolerate failures.

`Fenix_Comm_Add` can be used to notify Fenix about the creation of user communicators.

`Fenix_Checkpoint` performs a checkpoint of the specified element identifier, which has been previously initialized by `Fenix_Checkpoint_Allocate`.

`Fenix_Finalize` is the basic operation to terminate Fenix.

In terms of interface design, Fenix cannot take advantage of existing ideas about application-level checkpointing interfaces such SRS [149] or SCR [112]. The semantics of these are

richer in aspects not needed by Fenix, such as (1) no need to provide SRS's value restarting functionality because this is automatically done by Fenix upon failure, or (2) unlike SCR, Fenix creates checkpoints internally, and so operations such as `SCR_Complete_checkpoint` or `SCR_Route_file` are not needed. Furthermore, our interface provides additional features required by Fenix, such as the aforementioned registration of application communicators.

4.3.2 Integrating S3D with Fenix

As previously mentioned, changes to S3D were required in order to integrate it with the Fenix prototype. These changes are outlined in the skeleton below.

```
1  allocate(yspc(nx,ny,nz,nslvs))
2  allocate(other_arrays)
3  call MPI_Init()
4  [...] ! Initialize non-conflicting modules
5  call Fenix_Checkpoint_Allocate(C_LOC(yspc),
6       sizeof(yspc), ckpt_yspc)
7  call Fenix_Init(status, FENIX_CHECKPOINT_NEIGHBOR,
8       PEER_NODE_SIZE, FENIX_RESUME_INIT, C_LOC(world))
9
10 if(status.eq.Fenix_st_survivor) then
11       [...] ! Finalize conflicting modules
12 endif
13 [...] ! Initialize conflicting modules
14 if(status.eq.Fenix_st_new)
15       call initialize_yspc()
16 endif
17
18 do ! Main loop
19       [...]    ! Iterate and update yspc array
20       if(mod(step-1,CHECKPOINT_PERIOD).eq.0) then
21             call Fenix_Checkpoint(ckpt_yspc);
22       endif
23 enddo
24
```

```
25 call Fenix_Finalize()
26 call MPI_Finalize()
```

In the original S3D code, the array allocation (lines 1,2) was done after initializing modules (lines 10-13). It was moved to before line 5 to allow the proper invocation of `Fenix_Checkpoint_Allocate` (line 5) to allocate *yspc* as the element to be checkpointed. Note that conflicting modules assume a global state among all the cores to be maintained, or are modules that must be initialized collectively, for example. These modules are noteworthy because upon failure they must be re-initialized. One example is the S3D topology module, which is initialized by all ranks to re-create communicators. Some modules require finalization prior to re-initialization (e.g., the S3D derivative module). This is done on line 11.

`Fenix_Init` (line 7) is configured to resume to the beginning of execution instead of to the last checkpoint, as detailed in Section 4.2.1. Therefore, after recovering from a node failure (which is automatically detected by Fenix at every MPI operation invocation), all ranks will return to line 7, line 10 being the first instruction executed. Survivor processes will then finalize conflicting modules (line 11). After that, all ranks will merge in line 13 to initialize the conflicting modules collectively and continue the simulation in the main loop. The *yspc* array is initialized in line 15 the first time the execution is started.

As there is no collective synchronization in the main loop of S3D, its instructions might be executed at different wall clock times by the different ranks. However, before and after line 19, all ranks would have the same logical time/state (independent of the imbalance between the ranks). In other words, the element to be checkpointed (*yspc* array) will have a well-defined, globally strong-consistent value when all ranks finish, for example, line 19. The important point is that this statement still holds when all of the different cores reach that point, regardless of if they reach it at different wall times (i.e., independently of any imbalance in S3D). As a result, a checkpoint done at the end of iteration would generate a strongly consistent state. In addition, we only request a checkpoint every `CHECKPOINT_PERIOD` iterations, as we will show in Section 4.4.3.

In the following code snippet we show the rest of changes to S3D. Immediately after creating each derived communicator inside the topology module, `Fenix_Comm_Add` is called.

```
call MPI_Comm_split(gcomm, py+1000*pz, r, xcomm)
call MPI_Comm_split(gcomm, px+1000*pz, r, ycomm)
call MPI_Comm_split(gcomm, px+1000*py, r, zcomm)
call Fenix_Comm_Add(xcomm);
call Fenix_Comm_Add(ycomm);
call Fenix_Comm_Add(zcomm);
[...]
call MPI_Comm_split(gcomm, xid, r, yz_comm)
call MPI_Comm_split(gcomm, yid, r, xz_comm)
call MPI_Comm_split(gcomm, zid, r, xy_comm)
call Fenix_Comm_Add(yz_comm);
call Fenix_Comm_Add(xz_comm);
call Fenix_Comm_Add(xy_comm);
```

The overall impact on programmability using Fenix is very low, requiring less than 35 new, changed, or rearranged lines throughout the S3D code.

In containment domains [38] terminology, the presented integration of Fenix within S3D can be seen as a top-level domain that isolates multi-node failures from job crashes. The inclusion of more finely grained levels to further isolate multi-node failures is left as future work.

4.3.3 A Holistic Example

The following listing is a simple example of usage of a parallel matrix-matrix operation, encoded in `operate()`.

```
int main(int argc, char** argv)
{
  double A[N][N], B[N][N];
  unsigned int step;
  MPI_Comm world;
  int n, m;
  Fenix_Status rank_status;

  MPI_Init(&argc,&argv);
```

```
10    Fenix_Array ckpt_step, ckpt_A, ckpt_B;
11    Fenix_Checkpoint_Allocate((void *) &step,
12          sizeof(unsigned int), &ckpt_step);
13    Fenix_Checkpoint_Allocate((void *) A[0],
14          sizeof(double)*N*N, &ckpt_A);
15    Fenix_Checkpoint_Allocate((void *) B[0],
16          sizeof(double)*N*N, &ckpt_B);
17    Fenix_Init(&rank_status,
18          FENIX_CHECKPOINT_NEIGHBOR, PEER_NODE_SIZE,
19          FENIX_RESUME_CHECKPOINT,
20          &world);
21
22    step = 0;
23    Fenix_Checkpoint(ckpt_step);
24    initialize(A, B, world);
25    Fenix_Checkpoint(ckpt_A);
26    Fenix_Checkpoint(ckpt_B);
27
28    do {
29          operate(A, B, world);
30          step++;
31          if (step % RATE_CHECKPOINT == 0 ) {
32                Fenix_Checkpoint(ckpt_A);
33                Fenix_Checkpoint(ckpt_step);
34          }
35    } while ( step < ITERATIONS );
36
37    Fenix_Finalize();
38    MPI_Finalize();
39 }
```

The basic functionality of the code presented is to operate with two huge matrices, previously distributed among all the nodes, initialized, and updated `ITERATIONS` times.

Lines 11-16 allocate the three elements that will be saved during the execution, `A`, `B` and

`step`. Lines 17-20 initialize the library, requesting a neighbor checkpointing with restarting from the last checkpoint. This concrete application does not need to use `rank_status`. After this, lines 22-26 initialize the variables and checkpoint them. Lines 28-35 emulates the main loop in which the calculations are done, and the variables are checkpointed every `RATE_CHECKPOINT` iterations. As it is assumed that the matrix `B` is not changed by the `operate` function, there is no need to checkpoint it after its initialization. Note that in this example, the failure detection with occur during a call to an MPI operation (possibly within `operate`) or during the next call to `Fenix_Checkpoint`.

Note that while the current prototype implementation of Fenix uses ULFM, the presented interface is independent of ULFM and could potentially be implemented using other fault tolerance proposals that may be incorporated in the MPI standard. Therefore, the conclusions of this chapter extend beyond the presented Fenix prototype implementation. See Appendix A for the current version of the Fenix API.

4.4 Empirical Evaluation

If applications must tolerate high frequency failures (on the order of 30-300 seconds) that involve only a subset of the machine (e.g., a node, a blade, a cabinet...), sources of fault tolerance overhead must be reduced. By leveraging the fact that the majority of the system will survive these failures, on-line recovery can be used to further contain the failure. To enable on-line recovery, data must survive multi-node failures. Data checkpointing is therefore used. Specifically, Fenix implements diskless [123], double in-memory, neighbor-based [155] checkpointing.

In this section, the capabilities and impact of a prototype implementation of Fenix are evaluated on the Titan Cray XK7 system, using different benchmarks as well as the S3D [33] combustion simulation. The correctness of implicitly-coordinated checkpointing as previously described is evaluated and shown to further reduce overhead by not requiring any synchronization (either explicit or assumed) from the different compute ranks..

The ultimate aim of the experimentation is to run an execution mimicking a future extreme-scale scenario, in which node failures occur with extremely high frequency (i.e., every 47 seconds).

4.4.1 Methodology

In order to run this experiment, the following needs to be determined: (1) the checkpointing implementation overhead and scalability, both in terms of data size and total core count, (2) the optimal interval between checkpoints, calculated using the Young formula [154] and validated with empirical results, (3) the scalability of recovery algorithms towards an increasing number of concurrent failures within a group of ranks, and (4) the weak scalability of the recovery algorithm. Finally, we are able to run the experiment surviving frequent node failures assuming different system MTBFs.

Failures, or crashes, are injected by sending simultaneous `SIGKILL` signals to the involved application and runtime helper processes of a node. As a result, both the processes and the MPI runtime become unavailable to other nodes (i.e., this is seen as a real failure in the vicinity).

In an effort to reduce the impact of performance variability, all of the experiments presented in this section have been repeated five times, at different days of the week and at different times of day.

Note also that all experiments assume that the future extreme scale system job scheduler will offer the ability of adding spare ranks to the current allocation dynamically, upon failure. This is why the run time of the extra spare ranks that are requested in the allocation is not accounted as part of the performance, efficiency or cost. This is being actively researched in projects like PMI-X [30]. In case this assumption is false, the possible improvements due to the calculation of the optimal number of pre-allocated spare ranks falls out of the scope of this dissertation. The following experiments leverage checkpointing as a vehicle for data resilience, but improving its overall performance was not the goal of this dissertation. As a result, the `memcpy` process was not optimized by the compiler. Therefore, the results regarding checkpointing are an upper bound on the execution time.

Benchmarks. Before evaluating the framework with the S3D application, we perform experiments through the use of four benchmarks: **Laplace** uses a finite difference scheme to solve Laplace's equation for a square matrix distributed over a logical square processor topology. **Heat** computes an approximate solution to the time dependent one dimensional heat equation ($\frac{\mathrm{d}H}{\mathrm{d}t} - K \times \frac{\mathrm{d}^2H}{\mathrm{d}x^2} = f(x,t)$) using the finite difference method to discretize the

differential equation. **Poisson** solves Poisson's equation in a 2D region using the Jacobi iterative method to solve the linear system. **Matadd** uses a simple embarrassingly parallel matrix-matrix addition kernel to implement the example shown in Section 4.3.3.

4.4.2 Determining Failure-free Checkpoint Cost

Data size test. In order to know how the different checkpointing algorithms behave at different data scales, we first tested the four benchmarks with several data sizes.

Figure 4.3 shows the results of the four studied benchmarks. These tests have been conducted using 512 MPI ranks. The results are the average of five samples of each benchmark, where each sample is composed of ten kernel iterations. Note that each kernel iteration uses data from previous iterations. For this reason, we specified a checkpoint at the end of each iteration, to save the resulting data of the kernel. The values shown in the figure are, therefore, the average of $5 \times 10 = 50$ iterations. The log-log graph shows the checkpoint time of all elements (arrays, structures, and variables) in one iteration vs the size of the data, per rank, used by each benchmark. Each sub-plot includes both the time of the checksum algorithm (squares) and the neighbors algorithm (circles).

As is shown and expected, both algorithms scale linearly with input data. Anomalies witnessed in matrix-matrix addition size 8 (checksum) or heat size 8 (also checksum) could be due to network overutilization during some, or all, of the samplings. Also, we observe that checkpoint time is independent of the benchmark, depending only upon the data size. Finally, the checksum algorithm is about an order of magnitude less efficient than the neighbor-based checkpointing algorithm. However, it is still valuable because it demands less memory from the compute nodes.

Figure 4.4 shows the checkpoint time when using the S3D application for different data sizes, ranging from 0.07 to 15 MB per core (1 to 240 MB per node). Note that aggregating over all cores for a large-scale production run, e.g. O(100K) cores, yields a large amount of data in the order of terabytes.

The bars represent the average among all checkpoints, all cores, throughout five repetitions, while error bars indicate variability (including minimum, maximum, first, and third

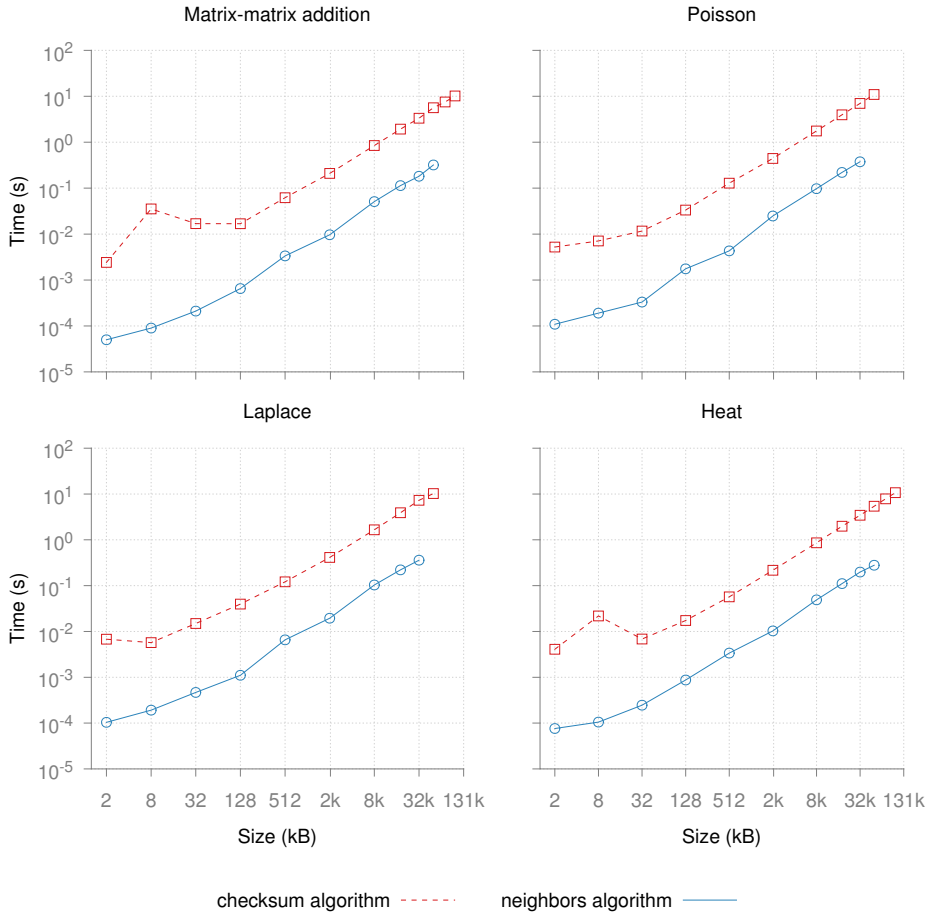


Figure 4.3: Effect of checkpoint size on checkpoint time for different benchmarks, and the two checkpointing algorithms. Both axes are in log scale.

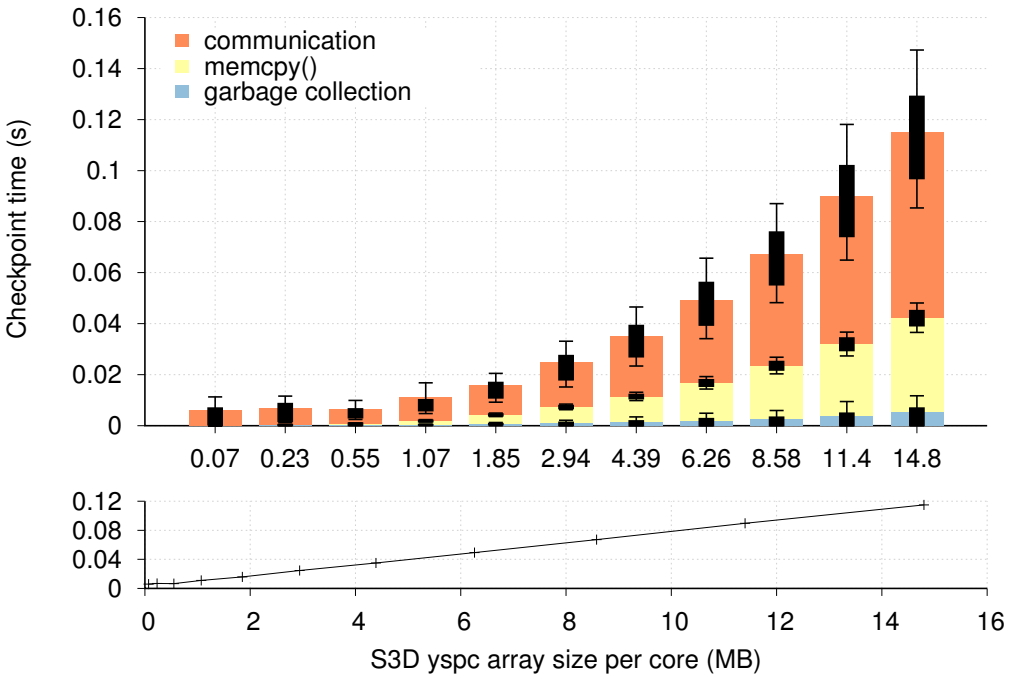


Figure 4.4: Checkpoint time for different data sizes (1000 cores). ©2014 IEEE (reprinted with permission) Gamell et al. [66].

quartile). The three different sub-bars show the three different processes that the checkpoint algorithm requires. Clearly, the communication cost dominates the execution. The lower plot in Figure 4.4 shows that the checkpoint time is linearly dependent on data size (for sizes greater than 1 MB/core), as expected.

The overhead caused by each array size strongly influences the choice of the size to be used in the rest of the experiments of this chapter – 50 grid points per core, which corresponds to 8.58 MB of the *yspc* array.

Studying weak scalability. Figure 4.5 shows how checkpointing scales to 250k cores as we increase total the number of cores while achieving similar average checkpoint time, sustaining a bandwidth of 16.8 TB/s in the test with a higher number of cores. This experiment has been performed using the S3D application. Again, the checkpoint procedure is dominated mostly by the transfer cost. As expected, the memcpy time remains constant throughout all executions, and the garbage collection cost is negligible. As this test does not need to tolerate failures, the MPICH distribution optimized by Cray is used instead of the ULFM prototype based on OpenMPI.

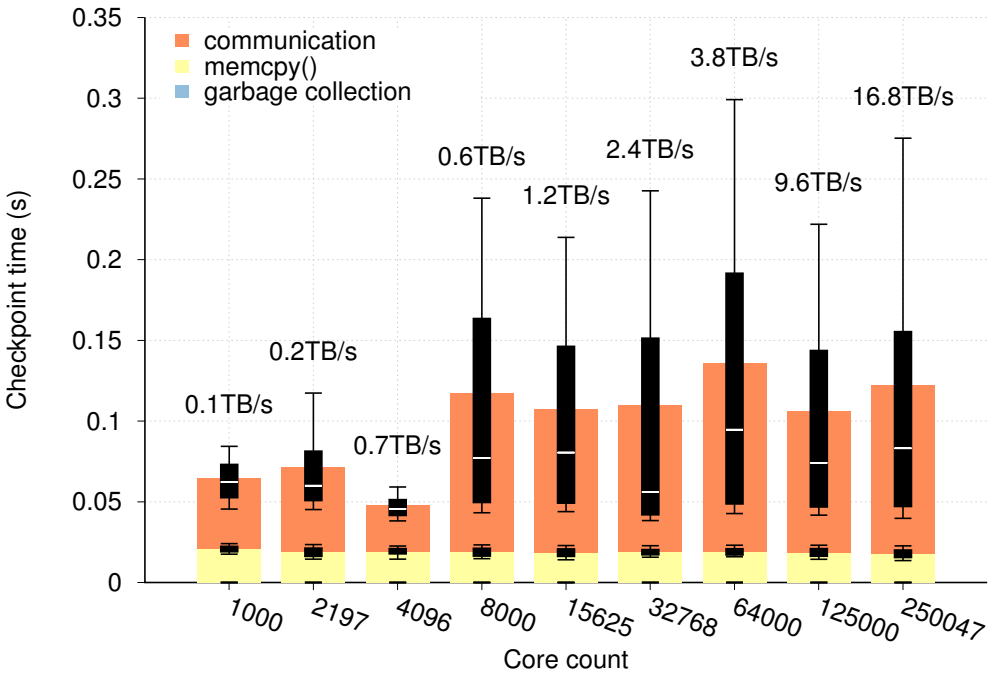


Figure 4.5: Checkpoint time for different core counts (8.6 MB/core). The numbers above each test show the aggregated bandwidth (the total checkpoint size over the average checkpoint time). ©2014 IEEE (reprinted with permission) Gamell et al. [66].

The lower communication time of the tests with less than 4k cores is due to the configured group size. In small tests it was set to 16 nodes, while in bigger ones was set to 96 nodes (the Cray XK7 cabinet size). As the group size is increased, messages must traverse more Gemini nodes[1] to reach the destination. The minimum of each test (the lower point on the error bars) is in all cases close to the third quartile. Furthermore, the median (the white line inside the error bar) is below 0.075 in all cases but in the 64k test. These observations indicate that 25% of cores finish the checkpoint process within a reasonably small time window and half of them take less than 0.075 s, while others take more time. As this section is not focused on the checkpointing process, no further analysis of this behavior is provided.

Assuming a linear relationship between checkpoint size and checkpoint writing time in ADIOS (which is the method used in production runs, as explained in Chapter 3), the production run's checkpoint time can be extrapolated assuming 8.58 MB/core. This would be translated to a 90-second checkpoint write overhead and a 72-second checkpoint read overhead, a 750-fold increase in the checkpoint time, compared to 0.12 s with 250

thousand ranks obtained with Fenix, see Figure 4.5. Note, however, that these experiments occurred on December 2013 without the newly installed ATLAS filesystems on Titan and, in January 2014, ATLAS is advertised to increase aggregated bandwidth to 1 TB/s [100]. Assuming this theoretical bandwidth is all used for checkpointing, and considering the same 8.58MB/core *yspc* vector, the writing process would take $(804GB)/(30GB/s) = 26.8$ seconds and a checkpoint loading time would be $(804GB)/(15GB/s) = 53.6$ seconds. This translates to 220× more time than the checkpoint time of 0.12 s with 250k ranks by using in-memory checkpointing. Regarding data recovery time, the Fenix implementation only requires the transfer of the checkpoints to the failed nodes, a process that can be expected to have the same overhead as the checkpoint operation. Compared to other studies such as CRUISE [125] – an extension of SCR [112]– the presented implementation is slower. This is mainly due to the fact that Fenix have to send the checkpoint remotely in order to tolerate entire-node failures, while tests done with CRUISE [125] only store checkpoints in local main memory.

Other considerations. The cost of resuming the computation is not evaluated as it only involves a local jump back to the start of the `Fenix_Init` function, which execution time is neglibible, in the order of a few processor instructions.
Note also that in small experiments a sequential allocation within the machine by the Cray scheduler is not ensured. Therefore, the results can be assumed to be a worst case scenario, because the checkpoints have to traverse more nodes to reach its final destination than what is expected to be with large contiguous allocations [1].

4.4.3 Validating Optimal Checkpoint Rate

First experiences with checkpoint frequency. After understanding how the checkpoint time depends on the data size, we next analyze how the total time is affected by the number of checkpoints performed, since an increase in the checkpoint period (the number of actual code iterations between checkpoints) will result in a decrease in the number of checkpoints.

Experiments with the four benchmarks have been conducted using 512 MPI ranks and a matrix of 512×512 elements per rank, totaling 1 GB per matrix. Each benchmark uses

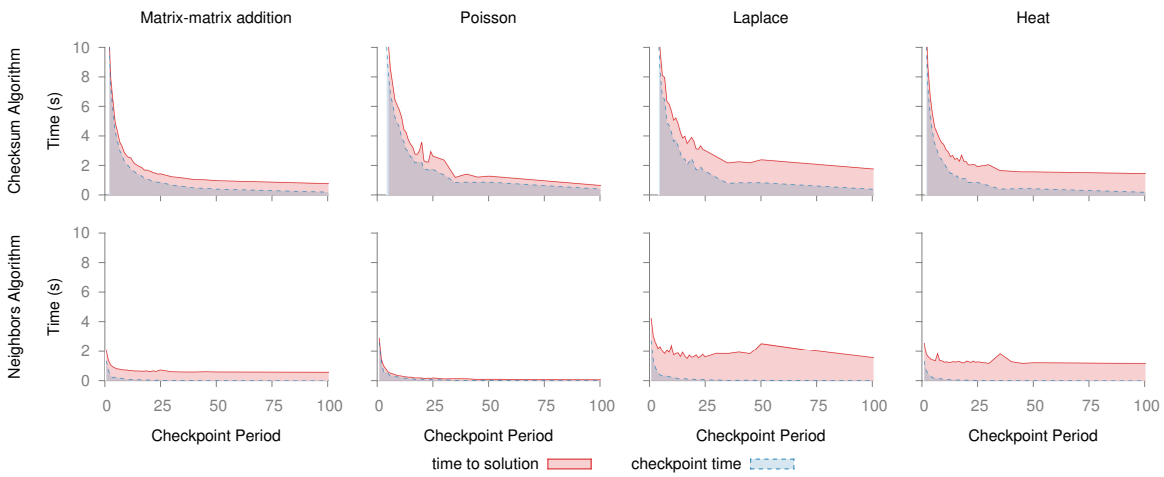


Figure 4.6: Effect of the checkpoint period (number of actual code iterations between two subsequent checkpoints) on the total time and the corresponding total checkpoint time for the different benchmarks and the two checkpointing algorithms.

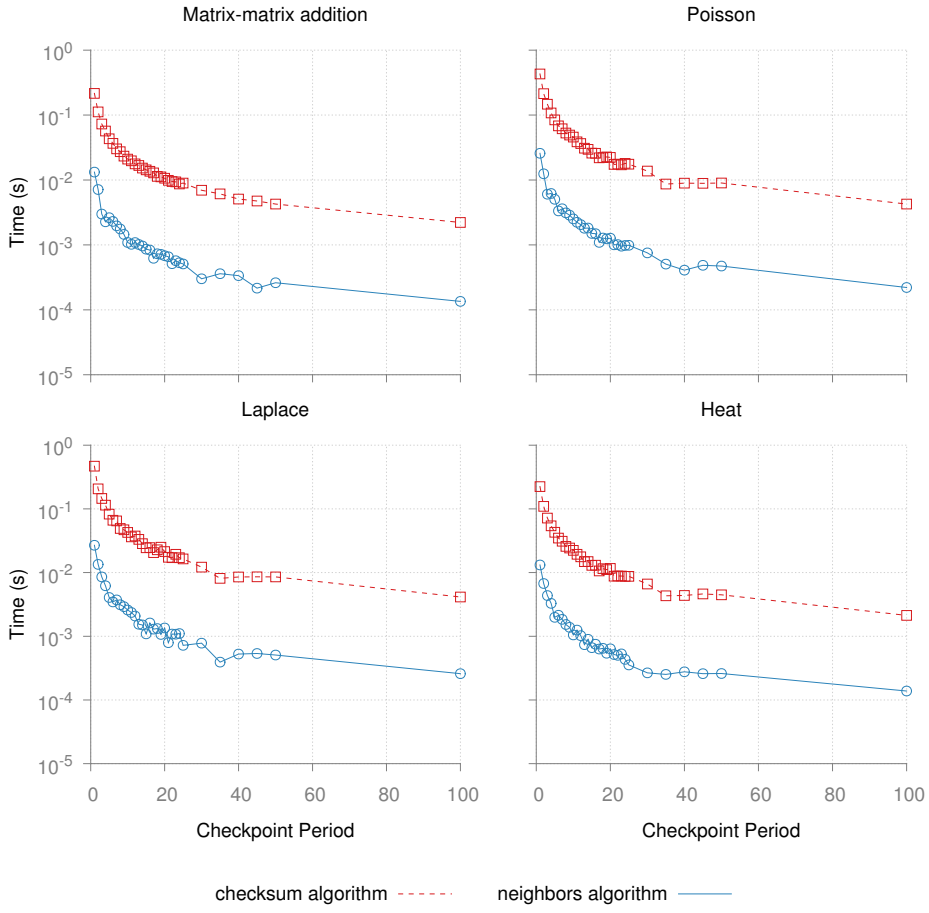


Figure 4.7: Amortized time cost of the checkpoint per iteration for different periods. y axis in log scale.

a different number of matrices. The results are the average of five samples, each with 100 iterations. Figure 4.6 shows checkpoint time (lower lines) and the total execution time (upper lines), which includes checkpoint time. Note that the difference between the lines is the benchmark computation time, which is not affected by the checkpoint period. To minimize the total time to solution, we want to decrease the checkpoint time as much as possible. Therefore, the longer the checkpoint period, the more efficient the execution will be. On the other hand, longer periods translate to increased recovery time, due to the work lost after a process failure. However, note that the cost decreases exponentially until it stabilizes at a period of about 25-40 iterations (depending on the benchmark) for the checksum algorithm and about 10-15 for the neighbors algorithm. This means that further lengthening the period will probably not help reduce the checkpoint overhead, but will increase the recovery time.

The results show that the neighbors method is more efficient, even at smaller periods, as it has been described above. For example, from the data in the figure, during a matrix-matrix addition over a period of 20 iterations, the application's total checkpoint time using checksum is 1.06 s, while when using the neighbors algorithm, the same test takes 0.06 s to perform the checkpoints. Also, with Poisson and a period of 50 iterations, the time to checkpoint with checksum is 0.89 s while it is 0.047 s when using neighbors.

These experiments have been conducted with a worst case assumption of a small computation time, which is much lower than the expected MTBF of future extreme-scale systems. In the more realistic scenario of an application with much longer iterations, the proportional impact of the checkpoint will be lower. This is shown in Figure 4.7, where we can see the amortized time of checkpoints per iteration: the incremental time added to each iteration in order to create checkpoints at a specific period.

Evaluating the optimal checkpoint rate. Young's formula [154, 137] can be used to determine T_C, the optimal interval between two consecutive checkpoints, depending on the MTBF of the system (T_F) and the checkpoint time (T_S). The checkpoint time has been determined in Section 4.4.2. As in the previous weak scalability test using S3D (see Section 4.4.2), checkpoint size is set to 8.58 MB/core, which leads to $T_S = 0.0748$ s in the case of 2197 cores (as shown in Figure 4.5). For a system with one million nodes, each

with an MTBF of 3 years, the overall system MTBF will drop to $T_F = 94.608$ seconds. Using second-order approximation for exponential distribution [154, 137], T_C is expressed as follows:

$$T_C = \sqrt{2T_S T_F} = \sqrt{2 \cdot 0.0748s \cdot 94.608s} = 3.76s \tag{4.2}$$

As the average S3D iteration time is 1.182 s with 50 grid points per core (over five executions of a failure- and checkpoint-free experiment on 2197 cores), T_C can be expressed as three S3D iterations rounded due to the fact that checkpoints are triggered by the application only at the end of iterations. Using the same procedure as in equation 4.2, the optimal number of iterations between checkpoints can be obtained assuming system' MTBFs of 47 seconds ($T_C = 2$) and 189 seconds ($T_C = 4$).

As suggested in [137], the proper usage of the formula, i.e. the correct parameter settings and the correct rounding of T_C from seconds to application iterations, needs to be verified. To do that, the total cost induced by a set of uniformly distributed, independent failures has been evaluated for several given checkpoint rates. Specifically, assuming an MTBF of 94 seconds, a Poisson distribution was used (rpois() with a seed of 10, $\lambda = 10$ from R suite, version 3.0.2) to obtain ten random possible failure timestamps within the 94-second time frame. The following timestamps were obtained: 12, 19, 24, 32, 41, 51, 61, 70, 78, and 91. Next, 10 different number of iterations between consecutive checkpoints had to be chosen. As the formula indicated frequent checkpoints, the concentration was focused on the smallest five (1 through 5 iterations). Also, to have an idea of the cost with lower frequency, another five were chosen to be disperse (10, 20, 30, 40, and 50 iterations).

For every checkpoint rate, the total overhead of fault tolerance was evaluated while injecting a failure to every chosen failure timestamp. The overheads induced in the resulting 100 experiments, each running 90 iterations, are represented in Figure 4.8. To determine which is the interval that offers the lowest overall overhead, Figure 4.9 shows the average of the overheads caused by the ten different failures, on each chosen checkpoint rate. Within the highlighted tests (2, 3, 4, and 5), checkpointing every three or four iterations offers the best overall solution, validating in turn the result from Young's formula.

Figure 4.10 shows the aggregated checkpoint time for different checkpoint rates, in a 128-iteration execution without injected failures. As expected, the aggregated checkpoint

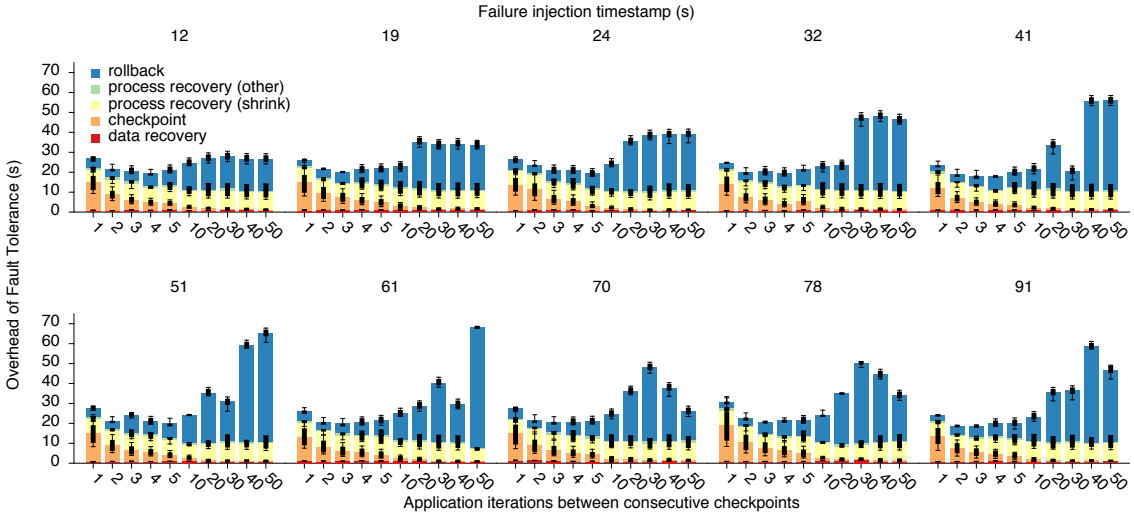


Figure 4.8: Overhead of Fault Tolerance for different checkpoint rates for several failure injection wall clock times (8.6 MB/core, 2197 cores, 90 iterations). ©2014 IEEE (reprinted with permission) Gamell et al. [66].

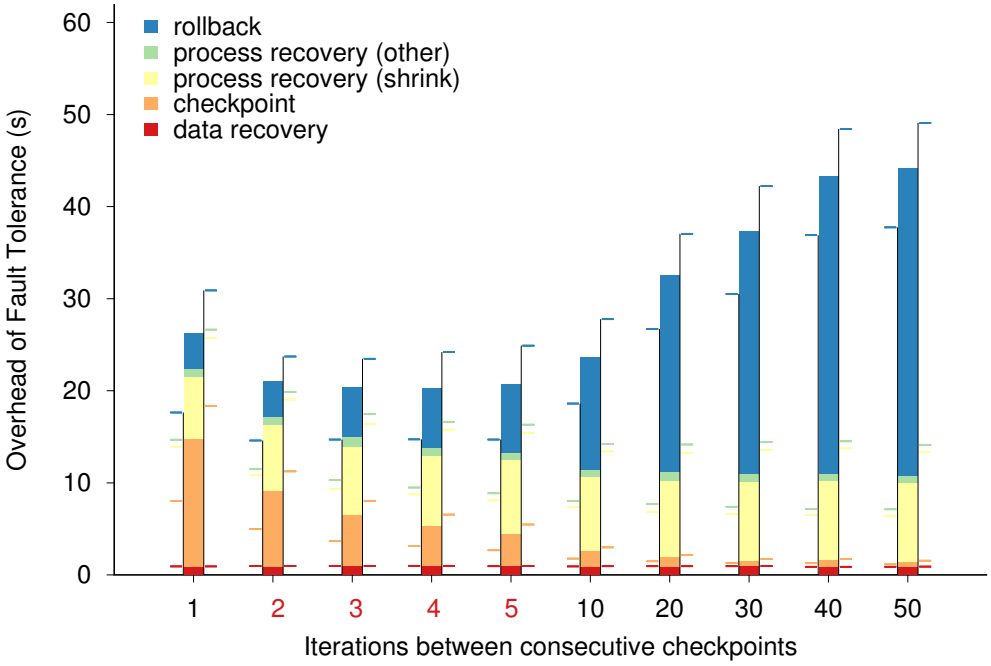


Figure 4.9: Average overhead for different checkpoint rates. Same test as Figure 4.8. ©2014 IEEE (reprinted with permission) Gamell et al. [66].

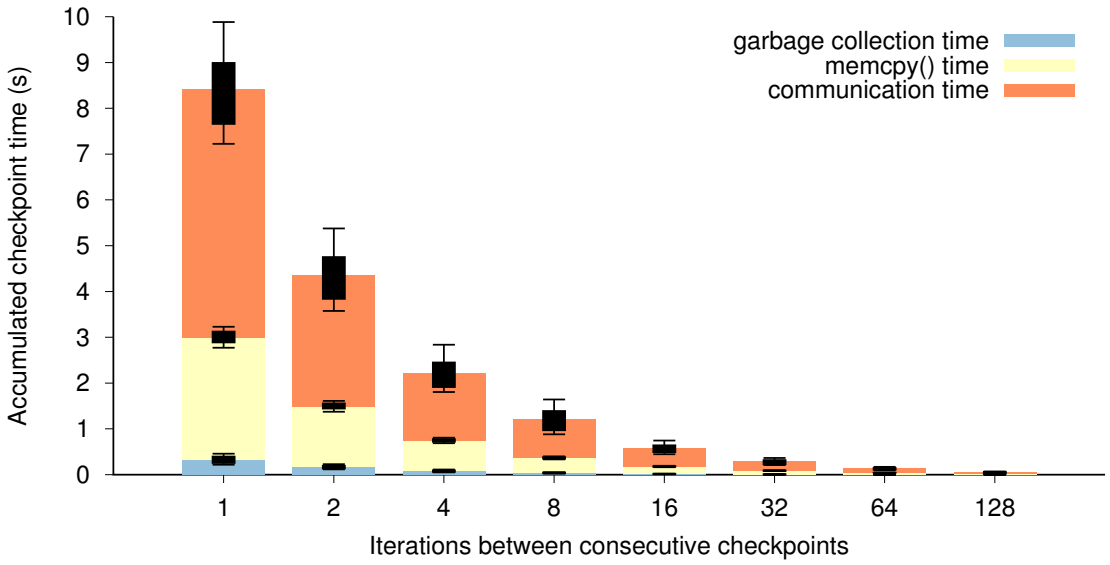


Figure 4.10: Accumulated checkpoint time for different rates (8.6MB/core, 128 iterations, 1000 cores)

time is perfectly proportional to the number of checkpoints.

4.4.4 Evaluating the Recovery Algorithm

In this section we evaluate the recovery portion of the Fenix library, as well as check that the implicitly coordinated approach produces reproducible results upon failure. To do this, we first execute multiple tests with increasing core counts using the four benchmarks. Then, we present a thorough evaluation of recovery scalability and the impact of the failure size to the recovery cost, using the S3D application.

Tolerating failures on-line. We first used the same four benchmarks, with the number of iterations increased to 10,000 for Laplace, 50,000 for Poisson, 20,000 for Heat, and 20,000 for matrix-matrix addition. In all cases, a checkpoint was performed every 500 iterations. As a result, Laplace performed a total of 20 checkpoints; Poisson, 100; Heat, and matrix-matrix addition, 40.

All runs were repeated five times, and the averaged results are shown in Table 4.1, with the exception of ULFM-enabled runs executing greater than 8192 MPI ranks. We were unable to run the 32k rank and 64k rank test injecting failures, as `MPI_Init` would fail to return when running with ULFM enabled. Therefore, all tests above 8192 ranks have

been done with ULFM disabled. Those tests are only useful to show the scalability of the neighbors technique with implicitly coordinated checkpoints.

We witnessed anomalies in the execution times for both 8192 rank runs of Heat and matrix-matrix addition, where the failure-free runs took longer than the corresponding single-failure runs. This is unexpected, and probably attributable to network congestion during the failure-free runs. We were allocated one partition on Titan, which we reused for successive benchmark runs. It is likely that these anomalies, clustered as they are particularly in performance time and partition space, were influenced by network congestion from concurrent jobs occupying the surrounding Titan partitions. Future testing will use multiple partitions, and variations in launch time, in an effort to normalize per-run anomalies such as these.

The total time-to-solution for the different tests increases with the number of cores, independently of the execution type: whether standard (sans our library), or with or without failures (using our library). Note that the checkpoint time cost of checksum algorithm increases linearly with core count. This is expected due to the collective operations used to calculate the checksum among all ranks. With the neighbors algorithm, however, we observe a constant cost with increasing core count; therefore this algorithm, combined with an implicitly coordinated approach, offers an scalable solution. Data recovery is slower when using checksum than when using neighbors, however, as data recovery using the neighbors algorithm is not impacted by the core count. The same observations can be made on all four benchmarks.

We see that process recovery takes more time when it involves more cores; it is expected that recovery time will be reduced in future versions of ULFM. In this dissertation we used the newest developer's version of ULFM, which in our current configuration does not reproducibly synchronize process recovery with high core counts; in order to guarantee process recovery, it was necessary to manually manage this synchronization using delay (sleep) periods.

We first measured benchmark execution without any additions for fault tolerance, using a non-fault-tolerant version of MPI (MPI column, Table 4.1). Also, we ran another set of tests activating fault tolerance only in the runtime (MPI+ULFM column). Those two tests

are useful to demonstrate ULFM's effect on the execution of regular, non-fault tolerant applications. We can see that, even though in the great majority of benchmarks this is negligible, there are some tests in which it differs. For example, in matrix-matrix addition the execution is faster using ULFM in all cases, while in Laplace it depends on the number of cores. In Poisson, ULFM penalizes the execution; while in Heat it is negligible. This may be due to the internal implementation of both the non-fault-tolerant and ULFM MPI versions.

In Table 4.1 we also analyze the overhead of our implementation. In the great majority of cases, the fault-free time-to-solution with our library is no more than ten percent greater than the non-fault-tolerant version's time-to-solution, and in many cases (especially considering the neighbors algorithm) the imposed overhead is less than five percent. This penalty may seem significant, but when weighed against the runtime penalty of an aborted global-checkpointing-based or even non-fault-tolerant execution –which could measure in hours– even a ten percent premium is a comparatively small price to pay to avoid a costly complete re-run in the event of a failure.

Note that the time-to-solution of both the checksum and the neighbors algorithms includes the checkpoint time cost, which is the factor that dominates the overhead induced by our library. The worst case example can be seen in Poisson, where the overhead is considerable because many data are checkpointed 100 times, and code execution time itself is extremely short, between 27 and 62 seconds for MPI+ULFM executions. In other cases we observe less than a ten percent overhead.

Increasing Failure Size. The following tests, performed using the S3D application, aim at studying the impact of the number of simultaneous failures on the recovery process. Given the same total number of nodes (2197 ranks) and the same problem size, failures are injected to an increasing number of nodes, ranging from 16 to 1024 cores. In all experiments the group size is set to 1024 so that failures are always injected inside the same group. Note that checkpoint times in all tests are between 95-104 ms, validating conclusion from Figure 4.5 about group size effect on checkpoint time from Section 4.4.2, which validates the conclusions.

The results of the experiment are shown on the left side of Figure 4.11, where similar total

	Cores	MPI	MPI+ ULFM	Checksum Algorithm						Neighbors algorithm					
				Time to solution		Checkpoint		Recovery		Time to solution		Checkpoint		Recovery	
				NF	F	NF	F	Process	Data	NF	F	NF	F	Process	Data
Matadd	512	115.507	113.897	124.986	136.862	8.649	8.599	8.462	0.564	119.981	135.377	0.554	0.427	8.508	0.055
	2048	148.750	144.489	158.224	169.712	10.415	10.516	10.541	0.772	146.733	166.944	0.553	0.385	10.542	0.052
	4096	173.144	166.701	182.465	204.697	11.702	11.703	16.099	1.025	177.517	200.637	0.563	0.407	16.395	0.054
	8192	221.935	212.680	366.974	268.787	26.922	14.600	36.924	1.286	191.604	231.071	0.558	0.438	36.812	0.056
	32768	213.073	-	-	-	-	-	-	-	230.171	-	0.491	-	-	-
	65536	241.473	-	-	-	-	-	-	-	255.235	-	0.502	-	-	-
Lapl.	486	150.206	145.330	155.736	169.297	8.558	8.899	8.460	0.579	149.909	168.795	0.474	0.343	8.541	0.104
	2027	171.339	177.061	193.388	212.550	11.171	10.800	11.595	0.790	181.967	202.978	0.481	0.366	10.330	0.112
	4098	191.592	203.696	219.012	236.997	13.667	13.494	16.210	1.609	203.869	224.110	0.497	0.359	16.377	0.106
Poisson	512	15.563	27.714	77.397	88.063	43.881	44.451	8.387	0.886	35.127	48.327	2.480	1.888	8.427	0.116
	2048	28.057	32.503	94.211	111.295	52.891	53.630	10.403	1.566	46.482	64.112	2.452	1.916	10.397	0.122
	4096	60.158	62.278	139.700	152.251	58.682	58.300	15.719	1.676	66.012	89.219	2.463	1.814	15.602	0.115
	8192	219.662	229.560	370.744	238.065	119.215	66.115	34.700	2.068	120.329	166.869	2.454	1.735	36.339	0.118
	32768	548.770	-	-	-	-	-	-	-	593.933	-	2.325	-	-	-
	65536	667.437	-	-	-	-	-	-	-	697.093	-	2.266	-	-	-
Heat	512	229.336	230.626	232.868	275.810	8.812	8.958	8.487	0.671	229.956	256.655	0.507	0.388	8.593	0.031
	2048	272.071	270.060	287.094	301.558	10.910	10.896	10.483	0.877	267.420	292.741	0.480	0.391	10.495	0.034
	4096	294.000	293.623	308.391	325.290	12.242	12.524	16.114	1.362	297.343	324.792	0.491	0.399	16.378	0.038

Table 4.1: Time (in s) results of four benchmarks. The MPI column is the time to solution for a regular execution of the benchmark with OpenMPI, with ULFM disabled, and without our library. The MPI+ULFM represents the same execution enabling ULFM features (without our library). NF represents failure-free solutions, F represents tests with one failure.

times can be seen (process recovery time T_P and data recovery time T_D) regardless failure size. It can be concluded that all parts of the recovery algorithm are scalable upon increasing failure size. Also, this result shows how Fenix's recovery algorithm enables tolerance to large sets of correlated failures with no extra cost.

Weak Scalability. After showing that the recovery algorithm scales when an increasing number of simultaneous failures occur, the effect of an increase of the total number of cores on process and data recovery needs to be analyzed. For this purpose, the experiment shown on the right side of Figure 4.11 is proposed. Three groups of ranks are set up per cabinet, resulting in a group size of 512 cores, since the cabinet size on Cray XK7 is 96 processors. The experiment was performed inducing a failure of 16 nodes (256 cores) per test, checkpointing every 16 iterations. Since the main goal of the test is to calculate the overhead of the recovery algorithm itself, not all the costs due to fault tolerance, any chosen checkpoint rate would have been valid. Comparing the overheads in Figure 4.11 (right), the following can be concluded: (1) the shrink algorithm does not scale (i.e. its overhead increases along with the number of cores), and (2) the remaining parts of process and data recovery algorithms scale perfectly.

Impact of spatial failure distribution. It is also interesting to study how the distribution of a failure within the group impacts the recovery time. The 4913-core test in Figure

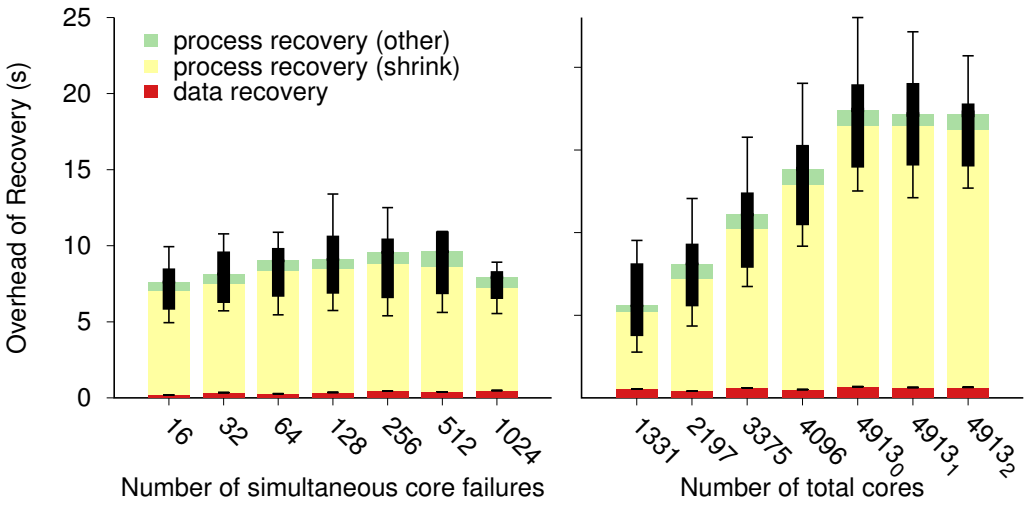


Figure 4.11: Recovery overhead. (Left) Simultaneous failures on increasing number of cores, 2197 total cores. (Right) 256-core failure on increasing number of total cores. The subindex in the 4913-core tests indicates a different distribution of failures within the 512-core group. ©2014 IEEE (reprinted with permission) Gamell et al. [66].

4.11-right shows three different distributions of a 16-node failure: 4913_0 injects failures in the lower 256 ranks of the 512-core group, 4913_1 injects failures in the upper 256 ranks, and 4913_2 injects 16-node failures randomly within the group. No impact of the distribution of the failure within the group can be observed.

Evaluating an alternative agreement algorithm. As previously noted, Figure 4.11 shows that the shrink algorithm implemented in the ULFM prototype used to perform the experiments above does not scale (i.e. its overhead increases along with the number of cores). The communicator shrink operation is based on the ULFM's resilient agreement algorithm. A new agreement algorithm, ERA, has been designed and implemented, which focus on the practical assumption that a process can return early in the agreement process [78]. In what follows, we re-evaluate the prior experiments to test the effect of the new agreement algorithm on the recovery process when compared to the baseline agreement algorithm (revision `b24c2e4` of the ULFM prototype). Figure 4.12 shows the results of these experiments in terms of total absolute cost of each call to the shrink operation. On Figure 4.12a we see how the operation scales with an increasing number of failures, from one node (16 cores) up to 64 nodes (1024 cores). We observe the drastic impact of

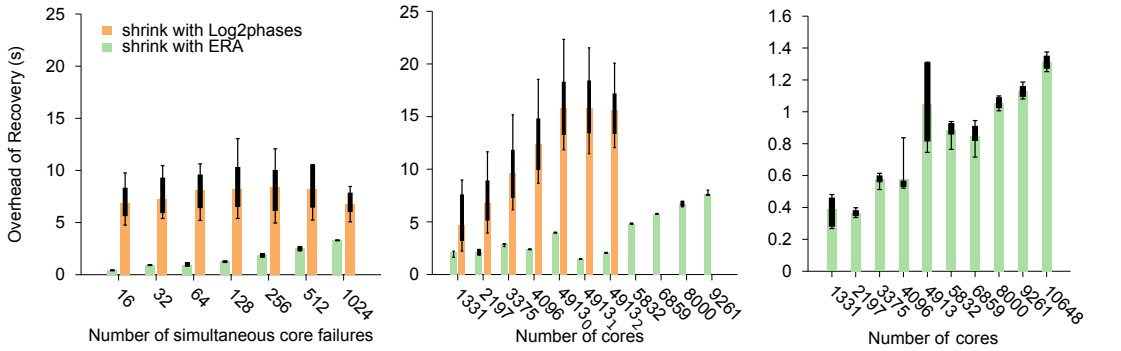


Figure 4.12: Recovery overhead of the shrink operation using the improved agreement algorithm (ERA), compared to the base algorithm (log2phases). In Figure 4.12a, simultaneous failures on increasing number of cores are injected, while fixing the total cores to 2197. In Figure 4.12b, 256-cores failures (i.e., 16 nodes) on increasing number of total cores are injected. The subindex in the 4913-cores tests indicates a different distribution of failures. In Figure 4.12c, 16-cores failures (i.e., 1 node) on increasing number of total cores are injected.

the new ERA agreement compared with the previous Log2phases algorithm, and the absolute time is clearly smaller with the new agreement algorithm, in all cases. By using the new agreement, however, the smaller the failure, the faster it is to recover. This is a highly desirable property, as described in Chapter 5, and cannot be observed when using the former agreement algorithm, in which case the recovery time takes the same amount of time regardless of the failure size. The results shown in Figure 4.12b represent executions injecting 256-cores failures using an increasing total number of cores. The new agreement is not only almost an order of magnitude faster, but scales to a number of processes not reachable before. It is also worth noting that the shape of the failure (i.e., the position of the nodes that fail, not only the number of nodes that fail) affects the recovery time with the new agreement algorithm, while this did not happen with the former. Finally, Figure 4.12c shows the scalability of the Fenix framework when injecting a 16-cores failure, which corresponds to a single node on Titan. As we can observe, the time to recover the communicator, while exhibiting a linear behavior, remains below 1.4 seconds when using more than 10,000 total cores. Clearly, we see a significant reduction in all cases.

4.4.5 Surviving Highly Frequent Node Failures

Goal. The results from the above experiments enable an empirical imitation of a future large-scale scenario (e.g. exascale), in which node failures might occur with much high frequency, such as one per minute in periods of high instability. The goal of this test is to show that on-line global recovery as presented in this chapter may be a solution for process/node failures towards these environments, at least by applications such as S3D.

Experimental set up. After determining the optimal checkpoint interval for the three scenarios (system MTBF of 47, 94, and 189 seconds), the experiments are set up to run on 2197 cores (problem size of $13 \times 13 \times 13$ with 50 grid points per core), injecting entire-node failures (16 cores). As done in Section 4.4.3, to determine the failure injection time a Poisson distribution with $\lambda = 47$, $\lambda = 94$, and $\lambda = 189$ seconds is used, respectively, (seed of 10 on the same version of the R suite). The test runs for 500 iterations and different overheads caused by all faults are measured, as well as the total time to solution. A summary of all these events for the three cases is shown in Figure 4.14. The time stamp of all events related to fault tolerance have been recorded as well, and are shown in the bottom side of Figure 4.13. Despite knowing the best optimal checkpoint (calculated in Section 4.4.3) each test was repeated with three different checkpoint rates, again validating the same conclusion as before (see Figure 4.14): for the 47s-MTBF test, checkpoint needs to be saved every 2 iterations, every 3 for the 94s-MTBF test, and every 4 for the 189s-MTBF test. Therefore, the discussion below is focused only on these three optimal tests.

Results. As seen in Figure 4.14, fault tolerance overhead come from three sources:

- (1) The recovery algorithm cost, which is split into (1.1) process recovery (`OMPI_Comm_shrink` and other costs) and (1.2) data recovery. Data recovery is clearly negligible, as expected from the algorithm design. Furthermore, process recovery is again negligible, aside from communicator shrink, which is the largest cost. However, it is dependent on the MPI implementation.

- (2) The overhead from re-execution of lost iterations, or rollback cost, is the second highest cost, only exceeded by the aforementioned shrink cost. Note, however, that the measured absolute rollback cost, about five seconds per failure in the 47-s

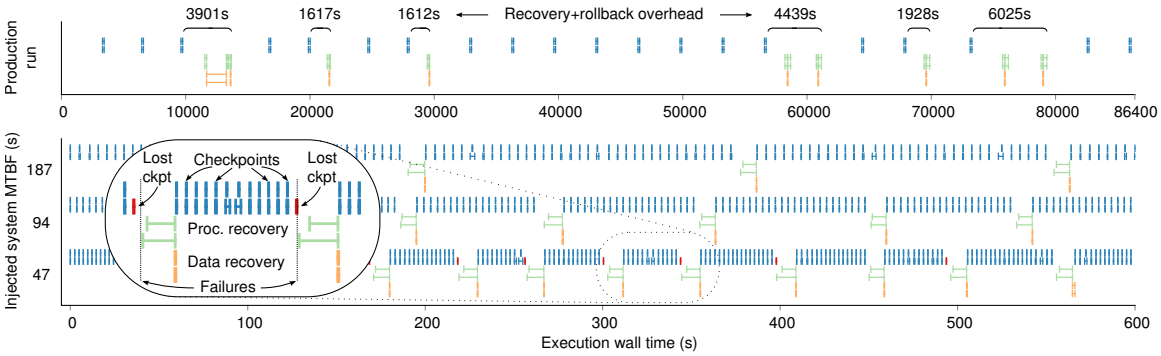


Figure 4.13: Checkpoint and failure timestamps on: (Top) production runs using 130k cores on Titan and (Bottom) the first 600 seconds of the high frequency node failure tests (8.6 MB/core, 2197 cores, 500 iterations, 16-core failure injection). Only one of the 5 re-executions is shown per test. Each test include six rows, organized by pairs. The pair's meaning is indicated in the zoomed area: from top to bottom, the first two indicate where the checkpoint occurred (in red, the non-finished checkpoints), the second two indicate the process recovery while the last two refer the data recovery. Within each pair, the top row shows the average time, while the bottom row shows the whole span througout the cores (i.e. the time between the first core begins until the last core ends). ©2014 IEEE (reprinted with permission) Gamell et al. [66].

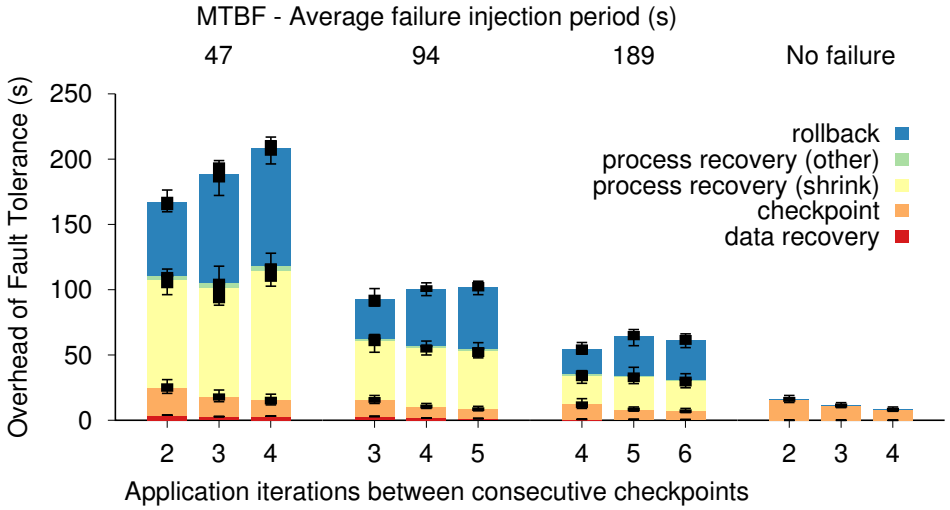


Figure 4.14: Overhead of continuously injecting failures at different periods, using different checkpoint rates. ©2014 IEEE (reprinted with permission) Gamell et al. [66].

MTBF test, is relatively small: ∼10% of 47. Its accumulated cost corresponds to below 9% over a failure-free, checkpoint-free test. It is also clearly smaller than the measured rollback cost in current production runs: 22.63% compared to a failure-free, checkpoint-free execution. Note that this scenario is even worse than what is expected towards exascale [49], where MTBF is predicted to be measured in minutes.

- (3) The aggregated checkpoint cost is the least significant cost of the three processes. In the worst case test (47-s MTBF), data is checkpointed on average ∼15 times between two consecutive failures, incurring an average cost of 1.05 seconds (2.23% of 47 s).

The second row in each test of Figure 4.13 (bottom) shows the time between the moment the first core begins the checkpoint process and the moment the last core finishes it. As this time is small for most checkpoints, and knowing that S3D does not have any synchronization directive within its regular execution path, it can be concluded that S3D is highly balanced. This conclusion applies for the great majority of iterations, but in some cases cores have started/finished the checkpointing process with a noticeable amount of imbalance, progressively resuming balanced behavior in subsequent iterations. These situations empirically demonstrate the real benefit offered by implicit coordination: no impact was produced by the checkpoint process due to this temporal imbalance. By accumulating the times in the second row a lower bound on the time to checkpoint for a blocking coordinated protocol can be estimated. The conclusions for the 47-s, 94-s, and 189-s MTBF test indicate, respectively, a cumulative blocking coordinated checkpoint of 76, 39, and 48 s which represent 3.4x, 3.0x, and 4.1x times more compared to using implicit coordination, as depicted in Figure 4.14. Note that in production runs shown in Chapter 3, where I/O costs are on the order of 55 s, coordination cost is insignificant. However, to checkpoint frequently coordination must be implicit to significantly reduce the overhead.

For a more finely grained perspective, specific events are analyzed individually as follows. The bottom part of Figure 4.13 shows all events related to fault tolerance. Specifically, for each test, it includes (top to bottom): (rows 1 and 2) when checkpoints happened, (rows 3 and 4) when failures happened –and process recovery time– and (rows 5 and 6) how much time data recovery took. Rollback time can be deduced from the last correctly finished

checkpoint (first/second rows in blue) to the beginning of each bar in row 4. Note that thanks to the well-balanced behavior of S3D, at most one checkpoint was lost per failure, which only happened in the 47-s MTBF test. The results also indicate how data recovery time is negligible in all cases; even in the recovery around second 560 of the 47-s MTBF test, in which data recovery took a slightly longer to finish. Also note that overall recovery time is very stable, taking in all cases the same amount of time.
The fact that lines in row 4 always begin before equivalent ones in row 3 indicate that the failure detection occurs at different times in every core of the system.

From a higher perspective, the tests are compared using the time to solution metric (i.e. from the start of the execution until the last core has finished) with respect to a failure-free and checkpoint-free execution. With failures injected every 189, 94, and 47 s, the total job run-time penalty is as low as 10%, 15%, and 31%, respectively, as shown in Figure 4.15. Chapter 3 shows that current production runs tolerate a cost of 31%, which is the same cost achieved simulating failures every 47 s.

As described in Section 4.4.4, the algorithm for shrinking the communicator has been dramatically improved. To estimate a lower bound on the total overhead we can assume that the shrink time is negligible and subtract it in Figure 4.14. The resulting overhead is reduced from 31% to 17%, from 15% to 8%, and from 10% to 6%. If we look at Figure 4.12a, we can observe that injecting 16-core failures in a 2197-core execution triggered a 6.85-second shrink with the former agreement algorithm and a 0.43-second shrink with the new agreement. Given that we see a 16-fold cost reduction of the shrink operation, it is safe to assume that the total overhead due to failures and fault tolerance has been reduced from 31% to 17.9%, from 15% to 8.4%, and from 10% to 6.2% for the 47-s, 94-s, and 189-s MTBFs, respectively, which is consistent with the lower bound.

4.4.6 Effect of the Checkpoint Size

Dongarra et al. [49] suggest that exascale systems are likely to have orders of magnitude less memory per core than current systems, estimating them in the order of tens of MB. Some estimations suggest that cores may have as low as 32 MB [17] of DRAM, assuming 100M cores [49], while other studies estimate that cores may have 250 MB [57].

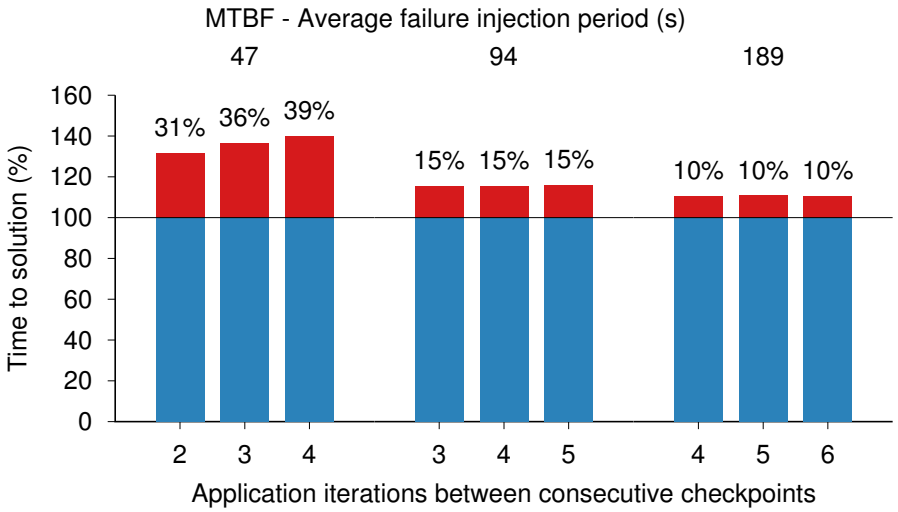


Figure 4.15: Overhead, as a percentage compared to a failure-free and checkpoint-free execution, of the different experiments of S3D with Fenix when injecting failures every 47, 94, and 189 seconds. The time-to-solution variability of the five repetitions done for each test is hidden in the decimal part of the percentages and hence, no error bars are required.

S3D, which uses relatively smaller amounts of memory per core, fits well with this architectural assumption. However, in cases where the assumption does not hold, i.e., exascale systems have larger memory sizes per core, an estimate of the overall overhead due to fault tolerance can still be drawn.

For example, for an application using 100 times the size utilized in Sections 4.4.3, 4.4.4 and 4.4.5, i.e., 858 MB/core, the 94-s MTBF experiment would require a checkpoint time of $T_S = 3.74$ s (experimentally evaluated on Titan), a checkpoint interval of $T_C = 26.61$ s, and a total overhead due to fault tolerance of approximately $T_C/2 + T_S \cdot T_F/T_C + T_P + T_D = 13.30 + 13.31 + 8.06 + 3.74 = 38.41$ s every 94.6 s. This corresponds to a 40.6% overhead, compared to the 15% for the S3D example assuming 8.58 MB/core. Similarly, using 85.8 MB/core (10 times the size used by S3D in this evaluation section), the 94s-MTBF experiment would require a checkpoint time of $T_S = 0.0748 \cdot 10$ s, a checkpoint interval of $T_C = 11.89$ s and a total overhead due to fault tolerance of approximately $T_C/2 + T_S \cdot T_F/T_C + T_P + T_D = 5.94 + 5.95 + 8.06 + 0.75 = 20.7$ s every 94.6 s. This corresponds to a 21 % overhead, compared to the 15 % for the S3D example assuming 8.58 MB/core. Therefore, the conclusions above would still apply in this case.

4.4.7 Evaluation Conclusion

These experiments empirically demonstrate how S3D, when augmented with Fenix, is able to tolerate exascale-level, high frequency process/node/blade/cabinet failures, leveraging high frequency checkpointing with a reasonable overall cost. For example, for a 94-second system MTBF, a 15% overhead has been observed and, when using the ERA agreement, this has been reduced to 8.4%.

4.5 On-line Recovery for Memory-filling Applications

Section 4.4 shows how on-line failure recovery, which enables applications to recover from failures without the need to stop the surviving processes, reduces response times, letting the application resume quickly after failures. Section 4.4 also shows that storing checkpoints in memory, instead of in the parallel filesystem (PFS for short), yields higher scalability, as the time spent to store or retrieve checkpoints depends only on the checkpoint size per computational resource, rather than the aggregated checkpoint size. It has been suggested that in-memory checkpointing, however, is not feasible for memory-hungry applications, since they typically occupy the entire memory on all allocated resources.

This section revisits on-line failure recovery and in-memory checkpointing, focusing on memory-hungry applications. We show that memory-hungry applications can often improve their performance and throughput by increasing their resource allocation to enable in-memory checkpointing. We model and simulate a wide range of application characteristics and machine features. We include an extensive sensitivity study covering a wide range of values for important characteristics, such as application memory usage, checkpoint size, and scalability, as well as machine reliability, process recovery time, and checkpoint storage bandwidth.

4.5.1 In-memory Checkpointing, Challenges and Benefits

As described in Section 4.2.2 double in-memory checkpointing stores a copy of the checkpoint in the local core's memory as well as in the memory of another core. In the worst case, this approach does not tolerate a failure affecting two cores. In practice it tolerates, with high

probability, all failures an application will suffer, if the chosen destination core is physically far away and shares no resources with the source, since the major fault mode in HPC centers is single node failures [108]. An advantage that this approach offers when compared to PFS-based checkpointing is constant weak scalability, i.e., the time to checkpoint depends on checkpoint size per core, not aggregate size, and is therefore independent of the total numbers of cores in the job.

In case of storing checkpoints in a centralized resource (PFS), however, an increase in the aggregated checkpoint size implies an increase in the checkpoint I/O time. In most cases, this is true even when using caching mechanisms as the final destination is still the PFS. For example, checkpoints created during a production run of the S3D combustion code using 130,000 cores of the Titan Cray XK7 at ORNL, and stored using state-of-art I/O technology (ADIOS) required about one minute to store for as little as 5.2 MB/core (see Chapter 3), i.e., an aggregate checkpoint size of 660 GB. Alternatively, storing 8.6 MB/core using double in-memory checkpointing requires less than 0.15 s (see Section 4.4), independent of the number of cores (tested up to 250,000 cores).

A major limitation of in-memory checkpoint storage is that a portion of the memory in all computational resources becomes unavailable to the application. Some applications, however, typically require the entire memory available in the job allocation. In those cases, in-memory checkpointing may be unusable as-is. This sections analyzes alternative usage modes that may enable in-memory checkpointing to be used by memory-filling applications. In particular, we study the costs and benefits of an increased job allocation or a reduced problem resolution.

4.5.2 Effect of Resource Allocation Increase

This section explores how application execution performance and/or throughput can be improved by (1) increasing resource allocation or (2) decreasing the problem resolution. Specifically, this section studies how the costs of fault tolerance are affected by different application characteristics and different machine features. In order to do so, we develop an application execution model and implement a simulator based on the model. We then use simulated executions of an application, injecting failures at different times. Our study

aims at understanding the impact of variations in the checkpoint storage destination and bandwidth, machine reliability, process recovery time, as well as application scalability, memory usage, and checkpoint size. For this effort to be realistic, we use the behavioral analysis of current production-level executions presented in Chapter 3.

It is assumed that applications do not require the entire machine to fit the problem size, but can run using only a portion of the machine.

Our results show that, in most cases, an allocation larger than the minimum allows not only a faster end-to-end time, but also a throughput reduction. This is due to the fact that larger allocations increase available memory, which can be used to create checkpoints faster and in a more scalable manner.

4.5.2.1 Goals and Assumptions

Aggregated time as a metric. Metrics such as end-to-end execution time, throughput, or floating-point operations per second (FLOPS) are commonly used to compare techniques or decide how to perform a particular computational task. Since an increase of throughput maximizes the use an HPC center and reduces the cost of running a job (in terms of allocation hours), this work focuses on the optimization of throughput, and we measure it as *aggregated time* (i.e., the product of wall-clock time and number of cores used). Aggregated times reported in this section ignore the impact of spare cores, as their count is negligible compared to the total core count.

IncMemStore. We refer to IncMemStore as the technique of storing increasing fractions of checkpoints in memory, made possible by increasing allocation size. As will be shown, IncMemStore may offer reduced aggregated time, which leads to a reduction of the end-to-end execution time.

Assumptions. Assuming that the work done by an application can be performed by an arbitrary number of cores (as long as the total problem size fits in memory) this study aims at finding the number of cores that maximize throughput (i.e., minimizes the total number of core-hours or process-hours spent to solve a problem). Each point in the figures in this section represents the ensemble average of 100,000 runs with the simulator, which uses a negative exponential distribution of failures with an expected time equaling the system

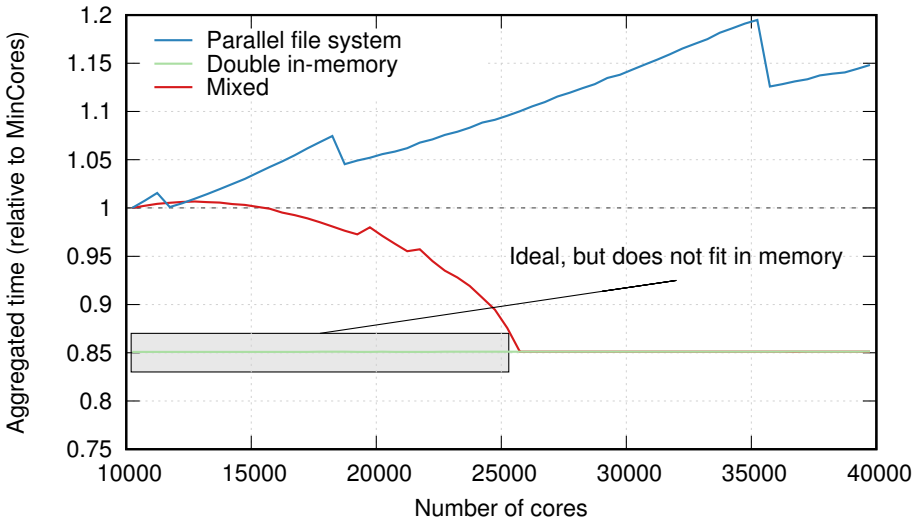


Figure 4.16: Comparison of three checkpoint storage methods. y-axis shows aggregated time, relative to MinCores.

MTBF. This means that different runs may experience different numbers of failures. In all the experiments, the Daly formula [41] is used to calculate the optimal checkpoint frequency (rounded to whole iterations), which depends on the time to create a checkpoint and the expected MTBF. Finally, in all cases we define the baseline configuration *MinCores* as that which exactly fits the entire application footprint, hence leaving no space for any in-memory checkpoint data.

4.5.2.2 Impact of checkpoint storage

Assume an iterative problem with perfect scalability that requires a total of 20 TB of memory–15 TB of which must be included in any checkpoint–and performs 16,000 iterations, each of which would require 12 hours if it were to execute sequentially in a single unit of execution. Figure 4.16 shows the aggregated time with varying numbers of cores of such an application, assuming each core has 2 GB of main memory, and can transfer (bidirectionally) messages to another core at 100 MB/s. Node MTBF is 1.5 years, and PFS bandwidth is assumed to be 12.5 GB/s. The node MTBF observed during the production executions described in Chapter 3 was of 2.5 years. We considered a shorter node MTBF to reflect the expected reduction in MTBF due to more aggressive frequency/voltage scaling at extreme scale; we study the effect of longer MTBFs in Section 4.5.2.4. In the figure,

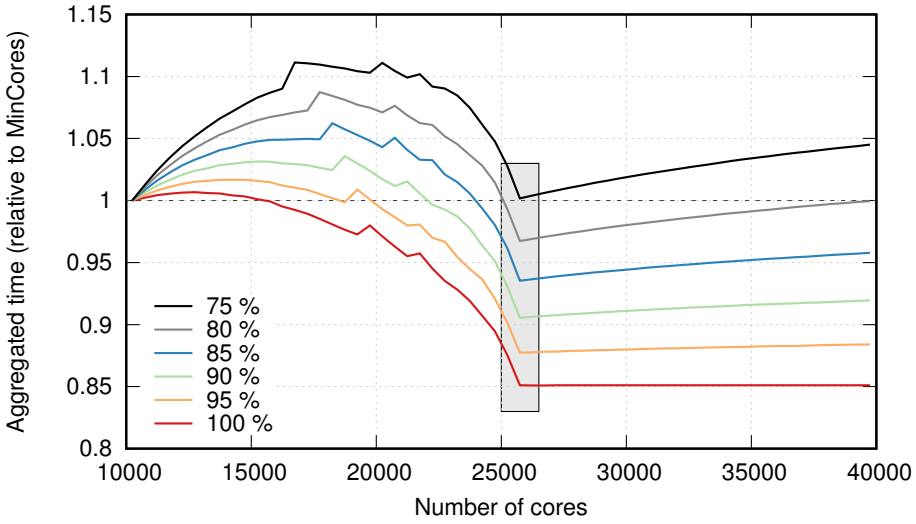
the aggregated time is relative to MinCores, which stores the checkpoints in PFS (left-most point of the *Parallel file system* line). The figure compares the three different checkpoint storage possibilities described in Section 4.1. Since the aggregated checkpoint size is fixed (the same problem is being solved), storing the checkpoint in PFS takes the same amount of time, independent of the number of cores. For that reason, an increase in the number of cores implies an increase in the aggregated time when using filesystem-based checkpointing. The saw tooth effect in the PFS-based curve is due to the fact that adding cores speeds up the computation, reducing the wall clock time of the program. When that reduction exceeds a threshold at a particular core count, one less checkpoint will be written, causing a drop in the aggregated time. These sharp drops, as well as the peaks in the mixed mode checkpointing curves, disappear when the number of iterations is increased tenfold. They are artifacts of the finite runtime, which makes Daly's formula, which holds for a steady state, non-optimal.

Alternatively, the cost of double in-memory checkpointing depends only on the size of the local part of the checkpoint and can be implemented with high scalability (see Section 4.4.2). Therefore, the total aggregated time of double in-memory checkpointing is the same, independent of the number of cores used, ignoring latency. Finally, note that by using a mixed approach, a small increase of core count above 10,000 actually damages throughput. This is because the part of checkpoint data stored in PFS still dominates the checkpoint cost and cannot be compensated by the small part of checkpoint data stored in memory. As more cores are added, however, the aggregated time quickly decreases below that of MinCores.

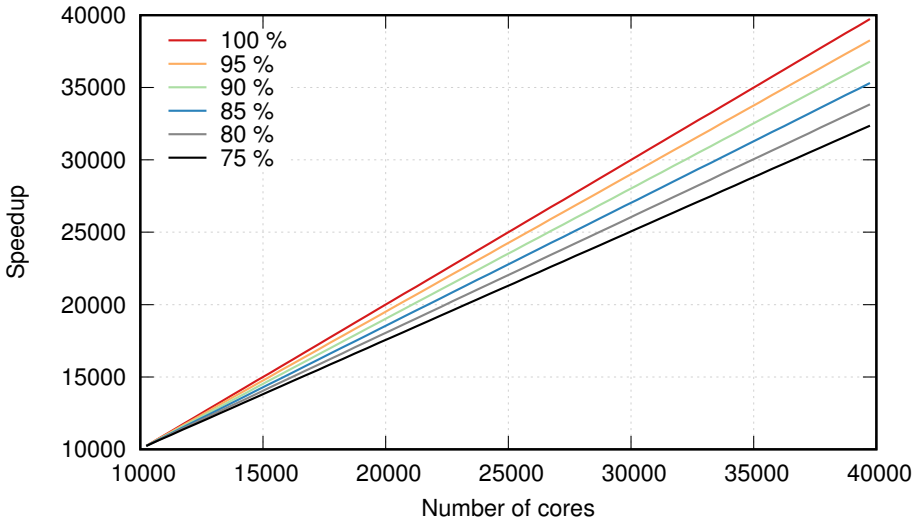
4.5.2.3 Impact of application scalability

In Section 4.5.2.2 we assumed linear scalability, e.g., doubling the number of cores results in halving the iteration time. Here we analyze how the application scalability affects the previous conclusions. To do so, we calculate the time T_{it} to complete an application iteration using P cores as

$$T_{it} = \frac{T_{it,min}}{S_f \cdot P/P_{min} + (1 - S_f)}$$



(a) Aggregated time relative to MinCores.



(b) Speedup of applications with different scalability factors.

Figure 4.17: Effect of different application scalability (modeled using the *scalability factor* metric as a percentage, ranging from 75% through 100%) and increased number of computational resources on the aggregated time.

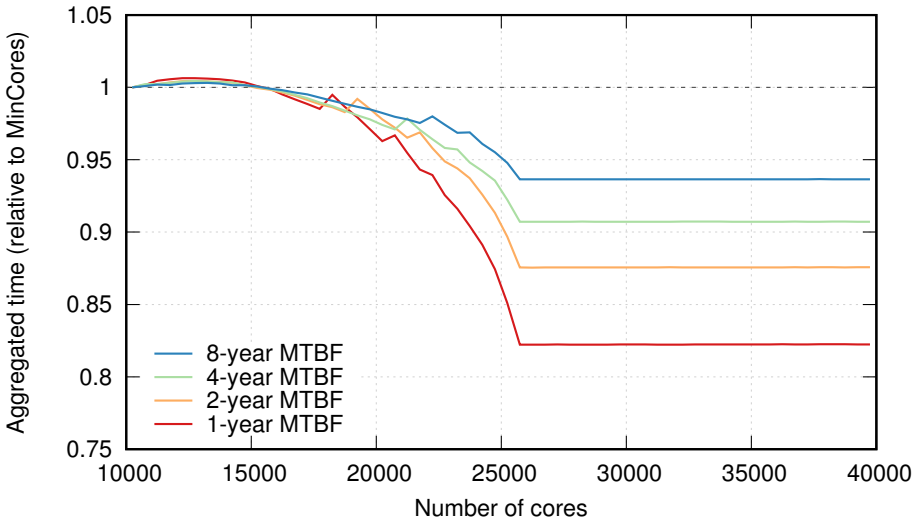


Figure 4.18: Comparison of four different node MTBFs. For each MTBF, a larger number of cores implies more failures in a given period of time. y-axis represents the aggregated time, relative to MinCores.

P_{min} is the minimum number of cores required to fit the application problem, i.e., $P_{min} = M/M_{pe}$, where M is the total memory usage of the application and M_{pe} the available memory per core. S_f is the scalability factor, which ranges from 0.0 (no speedup) to 1.0 (linear speedup). $T_{it,min}$ represents the time to complete one iteration using P_{min} cores. Figure 4.17b shows the speedup achieved when increasing P for applications with different speedup factors S_f. Similarly, Figure 4.17a shows the aggregated time when using the mixed checkpoint storage with applications with decreasing scalability factors S_f. Note that for each value of S_f the minimal aggregated time occurs when PFS stops being used (shaded box) and the entire checkpoint is stored in memory. Depicted applications with $S_f > 0.75$ benefit from IncMemStore, which optimizes aggregated time and, therefore, throughput. As a side effect, end-to-end execution time is also markedly shorter than for MinCores. Section 4.5.2.6 revisits scalability impact when considering other application features, studying a wider range of values for S_f.

4.5.2.4 Impact of the MTBF

When performing the previous simulations we assumed that each node had an MTBF of 1.5 years. We now analyze how node MTBF may affect the throughput advantages of IncMemStore. Figure 4.18 shows that the optimal number of cores that minimizes the

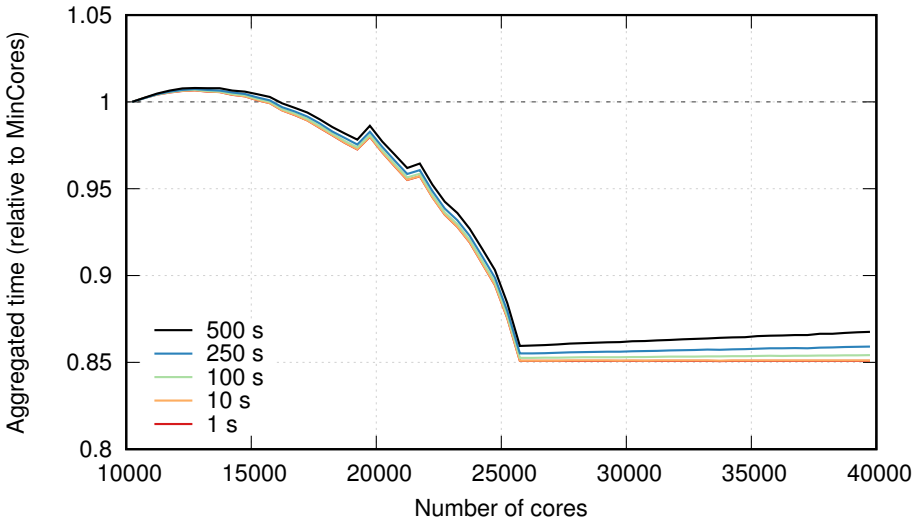


Figure 4.19: Effect of process recovery relative to MinCores.

aggregated time is independent of the MTBF, while the actual throughput increase does depend on the particular failure rate. The results in Figure 4.18 show that the benefit of IncMemStore over PFS increases for unreliable systems.

4.5.2.5 Impact of recovery time

Since the assumed MPI process recovery model is global, all cores participate after a failure. Therefore, a longer recovery time impacts the overall throughput gain when increasing the number of cores. In the previous simulations, a recovery time of 10s was used as an upper limit, based on the results of experiments performed on up to 10,648 cores on Titan that show a recovery time below 1.4s (see Section 4.4). Figure 4.19 depicts the impact of different recovery times, ranging from 1s to 500s, showing that while the impact is visible, IncMemStore is still beneficial in all cases. While in reality scaling effects are involved in the time to recover, we observe that in the range of system sizes considered (10k-100k cores), process recovery time can be considered constant since the largest component of its cost comes from application imbalance. Note also that the recovery time does not impact the optimal number of cores that minimizes the aggregated time.

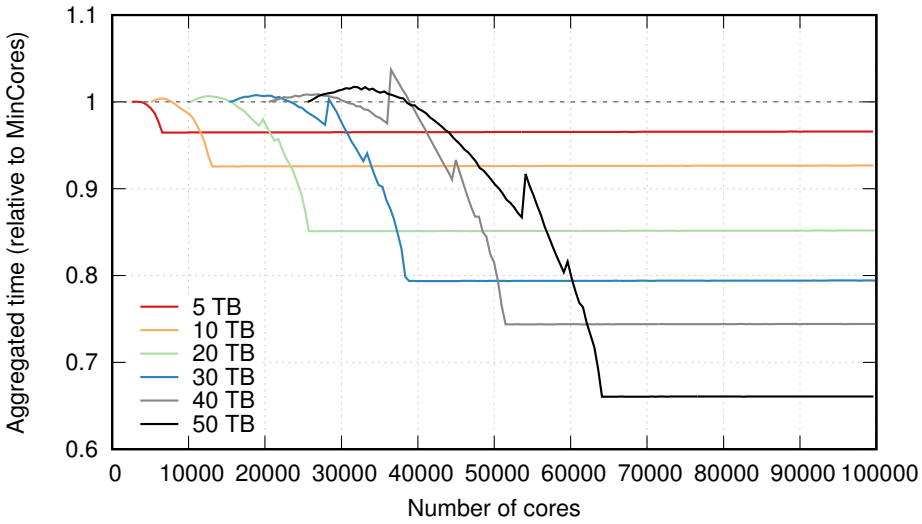


Figure 4.20: Effect of application memory usage running on an increasing number of cores, relative to MinCores.

4.5.2.6 Impact of memory usage and checkpoint size

Application memory requirements and checkpoint ratios (we define *checkpoint ratio* as the ratio of checkpoint size over application memory footprint) play a major role in the possible throughput gains of IncMemStore. Previous experiments assumed that the application required 20 TB of main memory, and that the checkpoints needed to include 75 % of application data (i.e., 15 TB). Figure 4.20 shows the aggregated time for an increasing number of cores for different application memory sizes, relative to MinCores. This simulation assumes a checkpoint ratio of 75% for an increasing application memory usage. The main conclusion is that the larger the application data size, the more beneficial IncMemStore is. In this example, applications that use a total amount of 50 TB of main memory can finish the simulation in about 66% of the aggregated time required by MinCores. At the other end of the spectrum, a smaller application that requires only 5 TB of memory can still benefit from IncMemStore, using about 97% of the aggregated time required by MinCores. Figure 4.21 shows a different scenario in which the application memory size is fixed at 20 TB and the checkpoint ratio ranges from 5% to 100%. As can be observed, the larger the checkpoint ratio, the more cores are required to minimize the aggregate execution time, and the more the latter can be minimized. However, larger checkpoint ratios allow lower aggregated time minima, relative to MinCores.

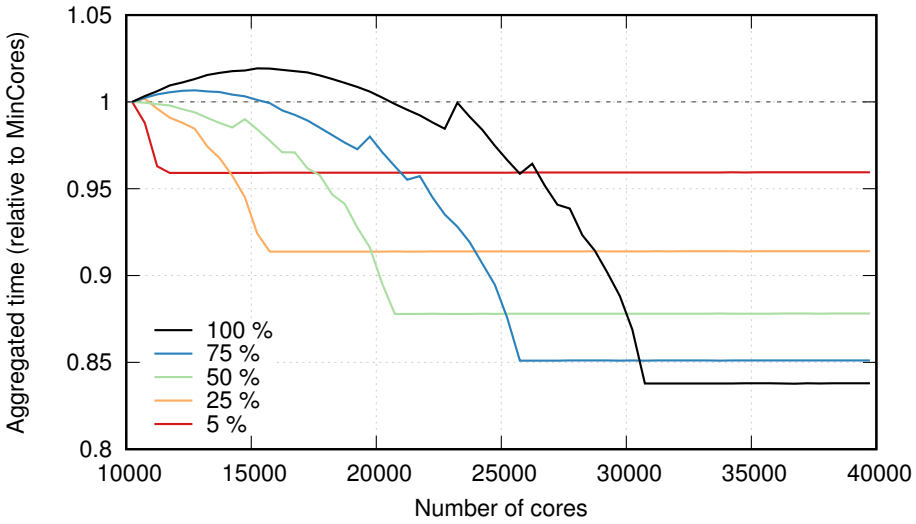


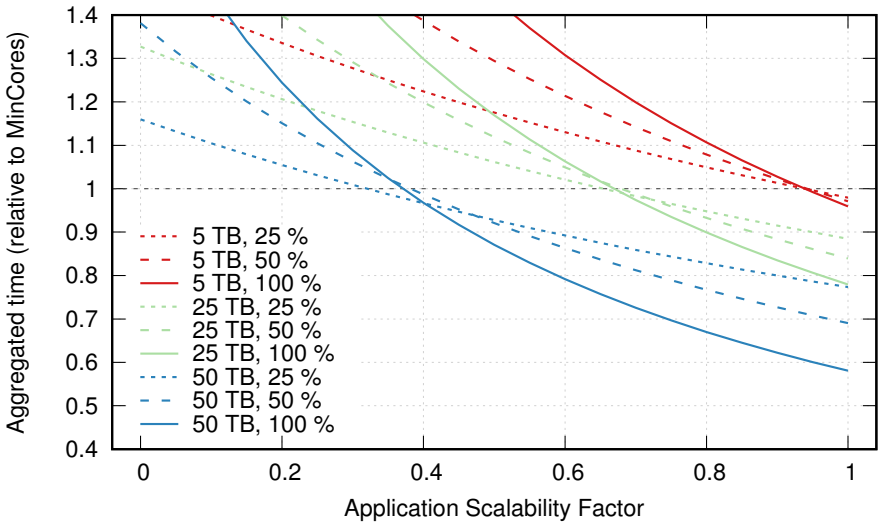
Figure 4.21: Effect of different checkpoint ratios on an increasing number of cores, relative to MinCores.

Figure 4.20 and Figure 4.21 show that both application memory usage and checkpoint ratios impact the optimal number of cores P_{mem}. This is expected since both characteristics determine the memory that needs to be available to store the entirety of the checkpoint in memory. In particular, the optimal number of cores P_{mem} can be found as

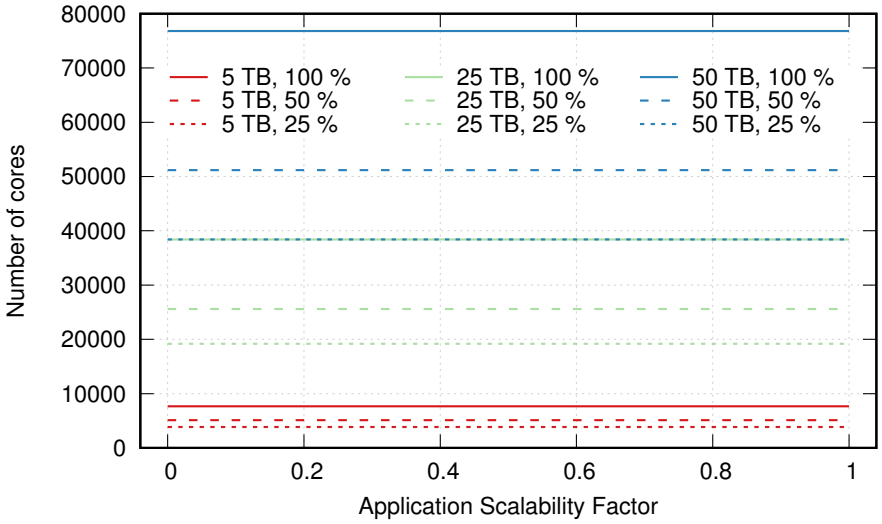
$$P_{mem} = (1 + 2 \cdot C_{ratio}) \cdot \frac{M}{M_{pe}} = (1 + 2 \cdot C_{ratio}) \cdot P_{min}$$

where C_{ratio} is the checkpoint ratio and is in the range [0,1] ([0,100%]), M is the total memory requirement of the application, and M_{pe} is the memory available in a single core.

Figures 4.22a and 4.23a plot the minimal aggregated time (relative to MinCores) corresponding to P_{mem}. Figure 4.22a shows how the application scalability impacts three different application memory sizes (5, 25, and 50 TB) and three different checkpoint ratios (25%, 50%, and 100%). The figure shows how bigger application memory sizes always lead to larger benefits from in-memory checkpointing. It can also be observed how, for applications that scale poorly, small checkpoint ratios benefit more from in-memory checkpointing than larger ratios. For applications that scale better this effect is reversed. Applications with smaller memory sizes (e.g., 5 TB) must have good scalability (e.g., scalability factors higher than 95%) to benefit from IncMemStore, while medium and large applications allow for a wider range of scalability factors (higher than about 65% and 35%, respectively) to



(a) Aggregated time relative to base test.



(b) Number of cores used.

Figure 4.22: Effect of increasing application scalability for different combinations of application memory usages and checkpoint ratios. Experiments labeled *"x TB, y %"* used x TB of main memory, and a checkpoint ratio of y %. Aggregated time for each scenario is relative to MinCores.

obtain benefits from IncMemStore. Figure 4.22b shows the optimal number of cores that are associated with Figure 4.22a. As described above, the scalability factor does not impact the optimal number of cores P_{mem}.

Figure 4.23a focuses on the study at a smaller granularity of the effect of application memory usage when using IncMemStore. While conclusions from this figure are consistent with previous conclusions it is important to note that applications that require more than about 37 TB of total aggregated main memory and have a scalability factor of at least 50% (possibly lower) can benefit from IncMemStore in all cases, regardless of checkpoint ratio. Applications with linear scalability benefit from IncMemStore regardless of memory usage or checkpoint requirements. Figure 4.23b depicts the optimal number of cores associated with each of the simulations shown in Figure 4.23a. As expected, P_{mem} increases linearly with the application memory usage as well as with the checkpoint ratio.

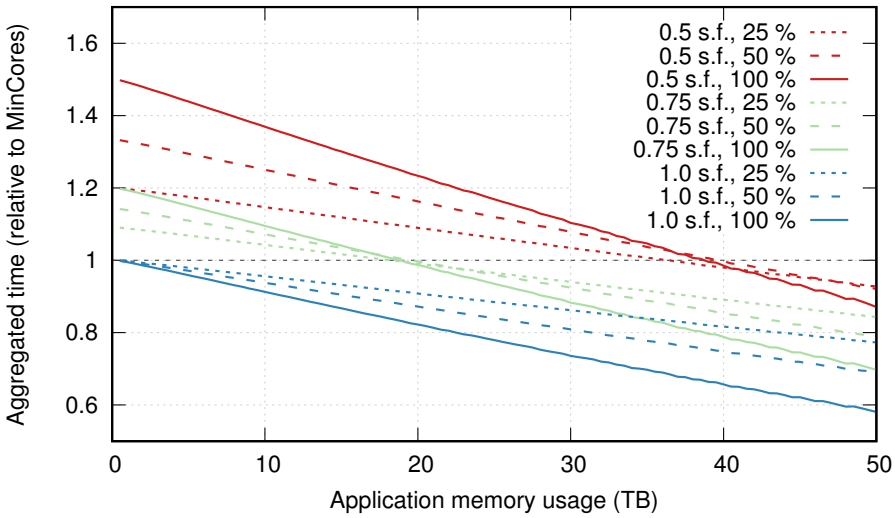
4.5.2.7 Impact of iteration time and number of iterations

We repeated the previous simulations with different total number of iterations (16,000 and 160,000) as well as different sequential iteration times (12h, 48h, and 180h), and no qualitiative changes were observed, aside from the smoothing of the performance curves when using higher iteration counts (see Section 4.5.2.2). This can be seen by comparing Figure 4.16, which simulates 16,000 application iterations, with Figure 4.24, which simulates 160,000 application iterations. Note how the curves are smooth in the latter. The same effect can be observed by comparing Figure 4.17a and Figure 4.25.

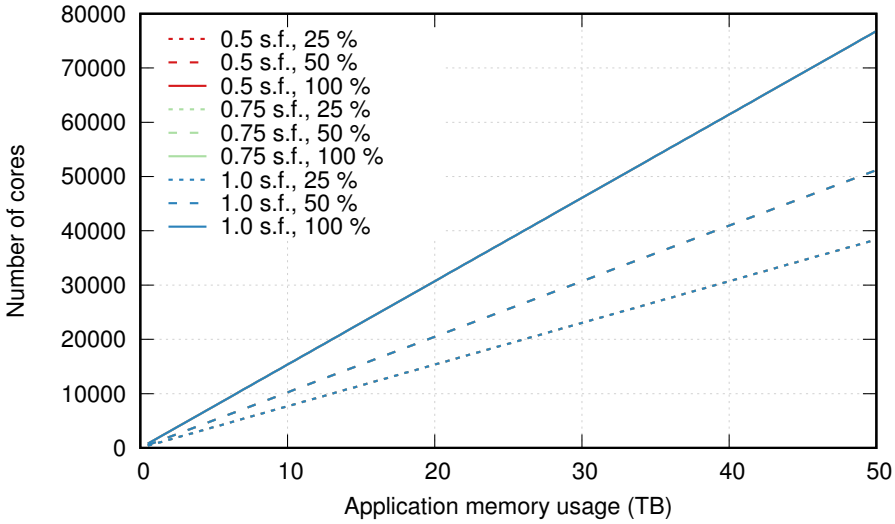
Therefore, the number of iterations and the iteration time do not affect the aggregated time ratio nor the optimal computation resources, regardless of the scalability of the application, the application memory usage, or the checkpoint ratio.

4.5.2.8 Impact of memory available per core

All previous simulations assumed cores with 2GB of accessible memory (based on Titan compute nodes). However, since it is unknown how much memory future systems will contain, it is important to study the impact of this architectural aspect on the throughput benefits of IncMemStore. Figure 4.26 shows the aggregated time when using P_{mem} cores,



(a) Aggregated time relative to base test.



(b) Number of cores used.

Figure 4.23: Effect of increasing application memory usage for different combinations of scalability and checkpoint ratios. An experiment labeled *"z s.f., y %"* has a scalability factor of z and a checkpoint ratio of y %. Aggregated time for each scenario is relative to MinCores.

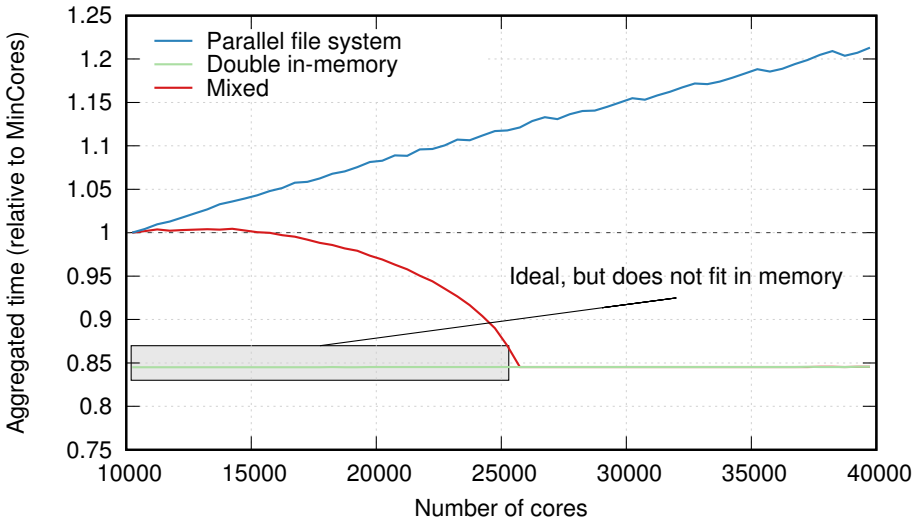


Figure 4.24: Comparison of three checkpoint storage methods, simulating a total of 160,000 application iterations. Compare with Figure 4.16, which simulates 16,000 application iterations.

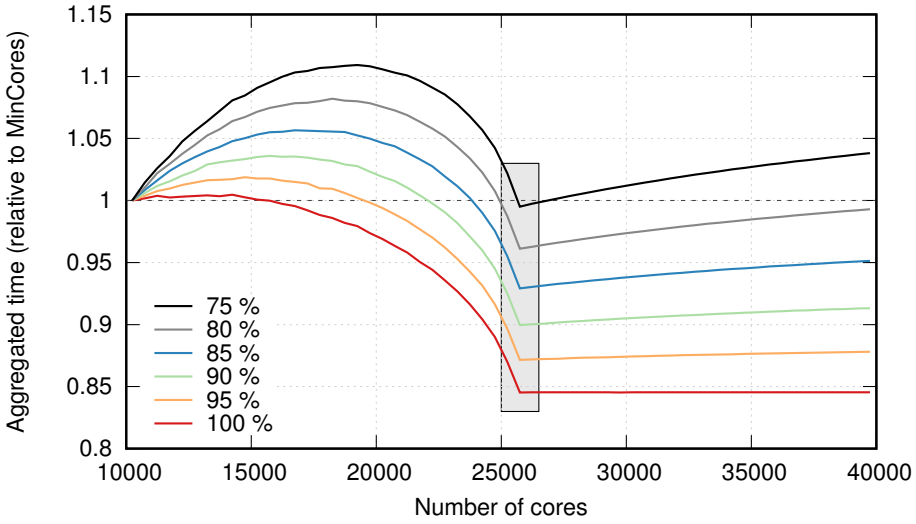


Figure 4.25: Effect of different application scalability and increased number of computational resources on the aggregated time, simulating a total of 160,000 application iterations. Compare with Figure 4.17a, which simulates 16,000 application iterations.

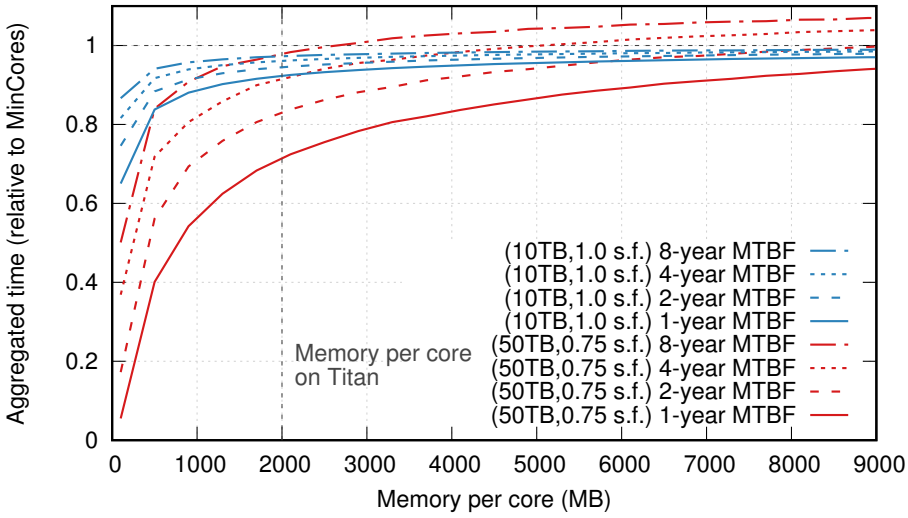


Figure 4.26: Effect of machine characteristics (memory size per core and node MTBF) on two applications. The label *"(50TB, 0.75 s.f.)"* represents an application with high memory requirements, while *"(10TB, 1.0 s.f.)"* represents an application with good scalability but lower memory usage. Number of iterations has been set to 160,000 for this test. Figure shows aggregated time relative to MinCores.

relative to MinCores, with increasing memory per core and different node MTBFs. Two applications are studied: large memory requirements and $S_f = 0.75$, and small memory requirements with perfect scalability (it is already evident from Figure 4.22a that poorly scaling applications with small memory requirements do not benefit from in-memory checkpointing, while poorly-scaling –$S_f \approx 0.35$– applications with large memory requirements do benefit). In both cases, the checkpoint ratio is 50%. A conclusion from this figure is that applications running on unreliable machines with small memory sizes per core, may obtain substantial benefits from IncMemStore. Note also that, when using unreliable machines (e.g., node MTBFs of 1 or 2 years), both applications present a reduction in the aggregated time regardless of the memory size per core, as previously shown by Figure 4.18.

The most important result, however, is that smaller available memory sizes lead to higher savings due to IncMemStore, specially for larger applications or less reliable machines. Exascale systems are expected to have smaller memory sizes per core than current systems, because they need to meet strict power limits, and hence need to employ processors with significantly more cores of lower frequency; even with a modest growth of memory per processor, memory per core is expected to decrease. This small-memory trend is consistent

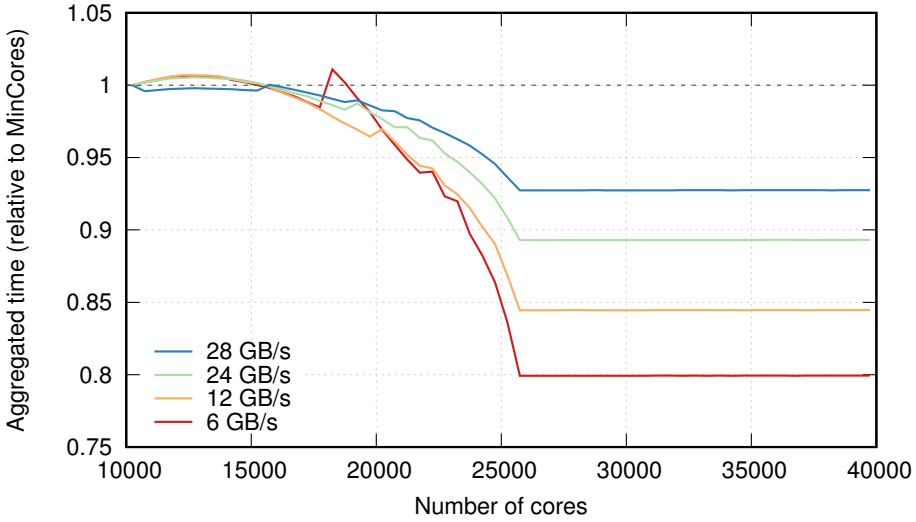
with current machines. For example, Titan, ranked first in the Top500 list of November 2012, has 299,008 cores and a main memory size of 598,016 GB (excluding GPU memory); about $598,016/299,008 = 2$ GB/core. Tianhe-2 (MilkyWay-2), ranked first in the Top500 list from June 2013 through November 2015, contains 3,120,000 cores and 1,024,000 GB of main memory; about $1,024,000/3,120,000 \sim 336$ MB/core. Finally, Sunway TaihuLight, ranked first in the Top500 list of June 2016, contains 10,649,600 cores with a total of 1,310,720 GB of main memory, which translates to $1,310,720/10,649,600 \sim 126$ MB/core. Consequently our conclusion indicates the strong viability IncMemStore offers for future machines. Note that even with current machine parameters (with higher memory sizes per core than expected at exascale), all simulated configurations benefit from IncMemStore by reducing the aggregated time.

4.5.2.9 Impact of checkpoint bandwidth

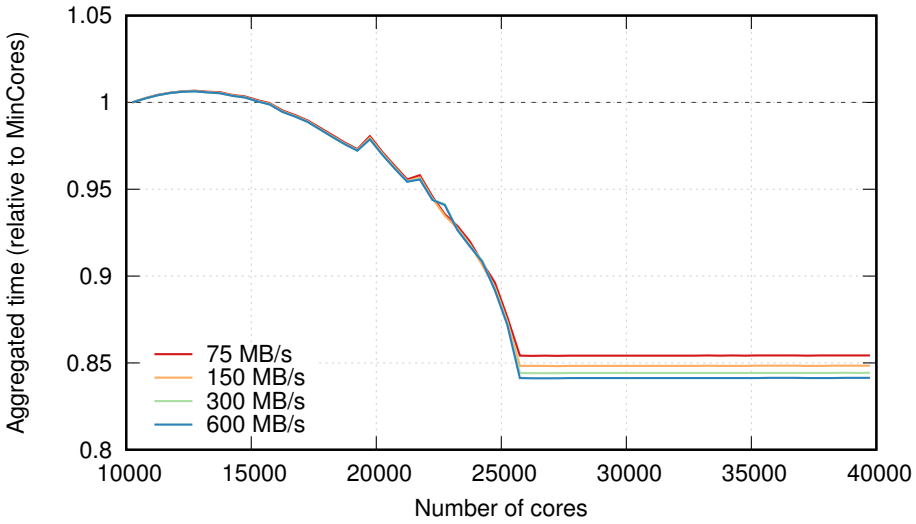
All experiments performed above assumed a PFS bandwidth of 660GB/55s = 12 GB/s, based on a duration of 55 seconds for checkpoints to be created by S3D production runs using state of the art I/O optimization techniques, and a total size of 660 GB. We used a representative bandwidth of 0.1 TB/s for double in-memory checkpointing on 1,000 cores (see Figure 4.5), which translates to a bandwidth of $0.1 \times 1024^2/1000 = 105$ MB/s per core.

Figure 4.27a shows how different PFS bandwidths affect the optimized aggregated time. In all the cases depicted, IncMemStore delivers savings in terms of aggregated time, while P_{mem} remains constant. Similar conclusions can be drawn from Figure 4.27b, which depicts the aggregated time when using four hypothetical machines with different node-to-node interconnect bandwidths that lead to different in-memory checkpointing bandwidth. This result shows how the potential benefits provided by IncMemStore are more sensitive to PFS bandwidth than to interconnect bandwidth.

Figure 4.28a shows how PFS bandwidth impacts the benefit of IncMemStore, using the same two application profiles as before. When using the large application, most of the configurations studied benefited greatly from IncMemStore. The benefits of IncMemStore for the small application are more modest, compared to the big application. However, the same conclusions apply. Figure 4.28b depicts the optimal number of cores that were

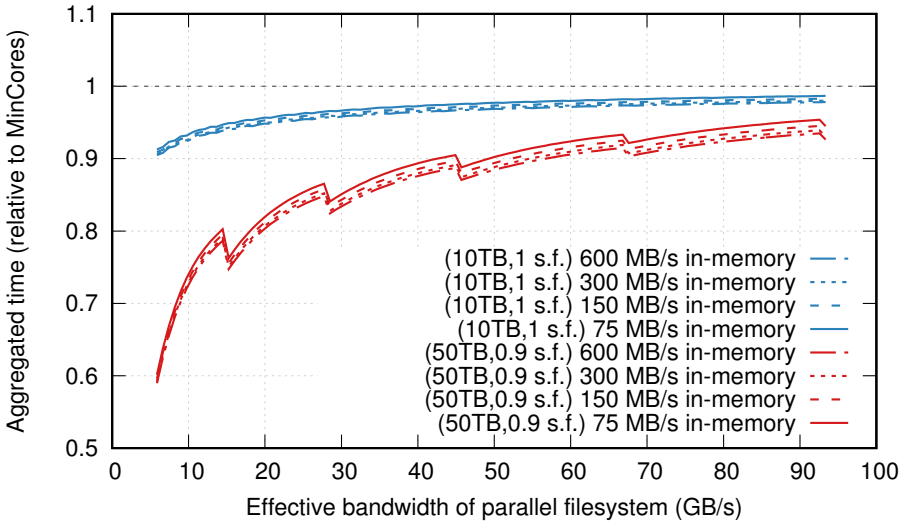


(a) Effect of PFS bandwidth on the aggregated time.

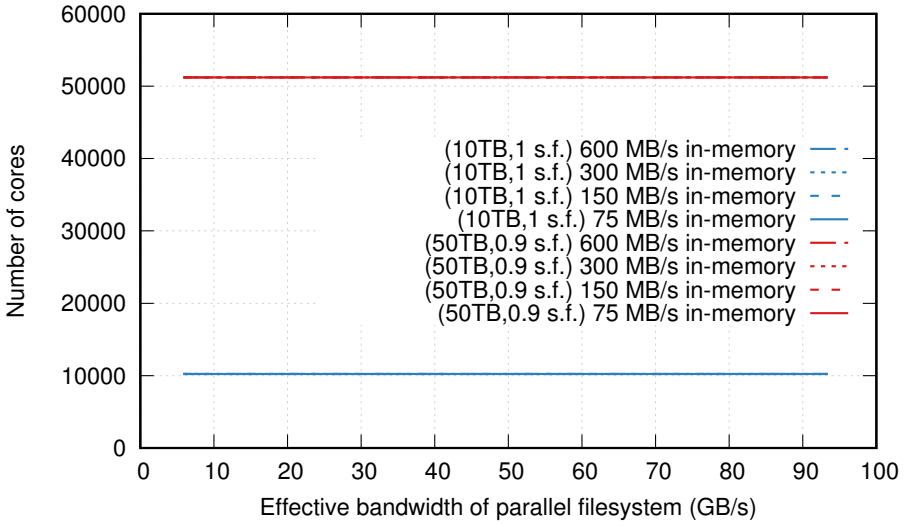


(b) Effect of in-memory bandwidths on the aggregated time.

Figure 4.27: Experiments to determine the impact of bandwidth on the effectiveness of IncMemStore. Depicted aggregated times are relative to MinCores. Experiments on current machines demonstrate an effective PFS bandwidth in the order of 12GB/s and an in-memory checkpointing bandwidth around 105 MB/s between any two MPI processes running on physically distant nodes, while allocating 16 MPI processes in each node. The node-to-node bandwidth is considered the bandwidth to transfer a single checkpoint from a core to its associated remote 'buddy', and, as experimentally shown, is considered to scale perfectly as more cores are added and transfer checkpoints simultaneously. The two studied applications represent the same as in Figure 4.26.

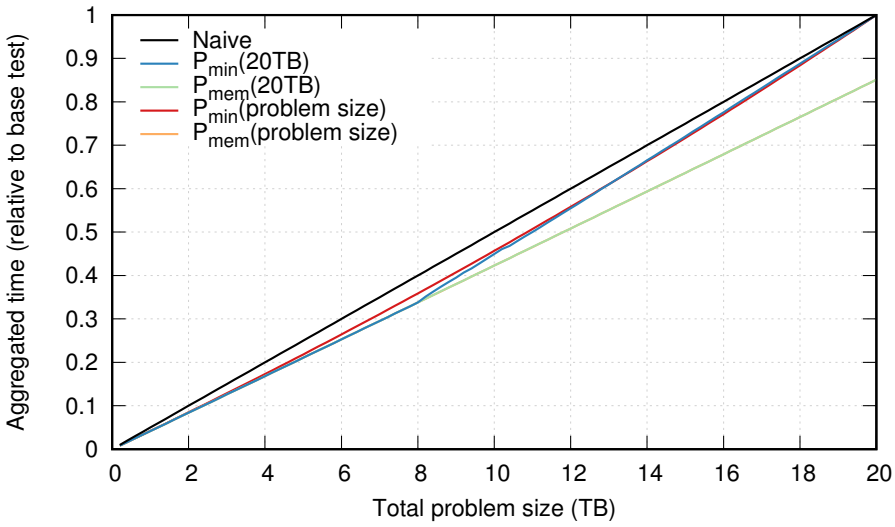


(a) Aggregated time with increasing PFS bandwidths for four in-memory checkpointing bandwidths.



(b) Number of cores used in the experiment depicted by Figure 4.28a.

Figure 4.28: Experiments to determine the impact of bandwidth on the effectiveness of IncMemStore (extension of Figure 4.27).



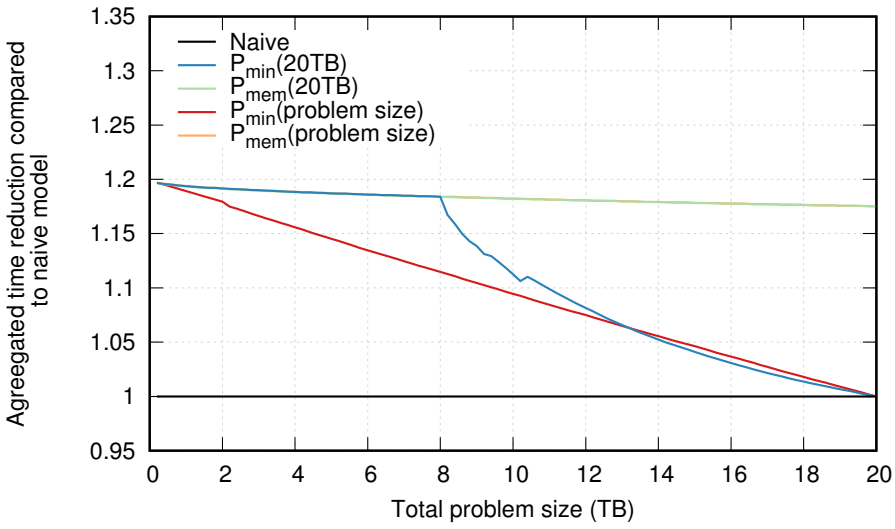
(a) Aggregated time relative to the base test.

Figure 4.29: Effect of different problem size reductions compared to a base test of 20TB scheduled on P_{min}.

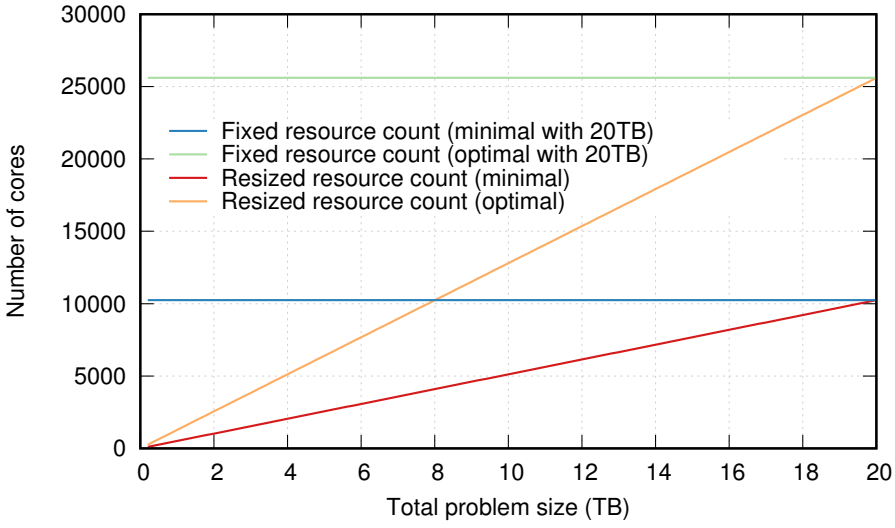
used when running the optimal experiments in Figure 4.28a. It can be observed that the checkpoint bandwidth does not impact the optimal number of cores, but, as stated above, the application size does. It can also be observed that the large and small case featured P_{mem} of 50,000 and 10,000 cores, respectively.

4.5.3 Effect of Problem Resolution Reduction

Previous experiments simulated a variety of scenarios maximizing throughput by using IncMemStore. The latter leverages memory pressure reduction achieved by distributing a fixed problem among a larger number of cores. However, there may be situations in which this approach is infeasible or undesirable, e.g., if the machine does not have enough cores or if the CPU-hour cost of larger allocations is higher than that of smaller allocations (tiered CPU-hour costs) and the difference is not compensated by the achieved aggregated time reduction. In those cases the user may want to reduce the problem resolution to obtain better throughput. Nominally, reduction of memory size by a factor of x and an algorithm with arithmetic complexity linear in memory size would also reduce aggregated time by a factor of x. Assuming PFS, we call this the *naive* model and label the corresponding curves in Figures 4.29a and 4.30a accordingly. However, if the reduced problem size is executed on the same allocation as the original problem, other benefits may be obtained:



(a) Aggregated time reduction relative to the naive model.



(b) Number of cores.

Figure 4.30: Effect of different problem size reductions compared to a base test of 20TB scheduled on P_{min} (extension of Figure 4.29).

(1) memory footprint is reduced, leaving more space for in-memory checkpoint storage, (2) checkpoint sizes are also reduced and, hence, are faster to create and more readily fit in memory made available by resolution reduction, and (3) execution time per iteration is reduced, yielding faster results and, hence, suffering fewer failures. Figures 4.29a and 4.30a study the combined effects of problem size reduction and in-memory checkpointing, using a checkpoint ratio of 75%. Figure 4.30a compares results to the naive model.

Experiments labeled $P_{min/mem}(s)$ set the number of cores to either P_{min} or P_{mem} as described in Section 4.5.2.6, corresponding to a problem size of s. We focus on the lines labeled P_{min}(20TB) and P_{mem}(20TB), which fix the number of cores used at 10,000 and 25,000, respectively. The lines labeled $P_{min/mem}$(problem size), which are plotted for comparison purposes, scale down or up the number of cores depending on the actual problem size, as has been done throughout Section 4.5.2. Note that, in all cases, combining in-memory checkpointing with problem size reduction achieves better aggregated times than when not leveraging in-memory checkpointing. When fixing the core count to P_{min}(20TB), an interesting point occurs with 8 TB (40% of 20 TB). At this point, the aggregated time is about 6.2% below the naive model prediction, as shown in Figure 4.29a. As can be seen in Figure 4.30a, the aggregated time consumed is around 1.19 smaller than given by the naive model.

Examples of applications that can directly benefit from this technique can be found in reduced-order modeling, which seeks inexpensive but robust mathematical representations from multiple low fidelity simulations (i.e., reduced problem sizes with lower resolution). The associated inaccuracies of these low-fidelity simulations are corrected through the projection from a small number of high (and more expensive) fidelity simulation outputs [28]. For example, a higher fidelity simulation may require 20TB and use a certain amount of aggregated time. Without using in-memory checkpointing, four lower-fidelity simulations of 5TB can be executed using the same amount of aggregated time. In the same case, when using in-memory checkpointing, we may be able to run almost five ($1.19 \times 4 = 4.76$) lower-fidelity simulations using the same amount of aggregated time.

Chapter 5

Local Recovery for Stencil-based Scientific Applications

This chapter presents the design, prototype implementation, and evaluation of local recovery approaches for certain classes of applications in FenixLR. Specifically, the feasibility of local recovery is studied 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 addition 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.

5.1 Overview

Global recovery. Chapter 4 introduced the technique of recovering from a multi-process failure in an on-line manner; that is, enabling ranks/processes that survived a process or multi-process failure to recover from it without disrupting them. Making all the processes aware of the failure, and, hence, part of the recovery, is, however, suboptimal. It is costly, less scalable and, in many situations, unnecessary. The goal of this chapter is to study how global recovery can be substituted by local recovery under certain assumptions that guarantee high efficiency.

Upon failure, the presented approach allows all processes in the system to continue working and only trigger the rollback of the failed rank, which has to be substituted by a spare resource or a re-spawned resource.

Local recovery in task-centric applications architectures. Task-centric application architectures, such as for example "bag of non-parallel tasks", are an ideal use case for local recovery. In these applications, each task is assigned one or multiple processing units

(e.g. processor cores) in the system, and many tasks are spawned in parallel throughout the system. When a process/node failure occurs, only failed tasks have to be restarted. All other tasks on the system do not have to be aware of the failure, since they are independent and do not directly communicate with each other. In order to avoid restarting each task from the beginning, periodic checkpoints can be taken during the execution of a task, and upon task recovery, the task can be restarted from the last checkpoint, in a local manner.

Local recovery for stencil-based parallel (MPI) applications. The aforementioned scenario, even if it is desirable and embarrassingly scalable, it is not realistic for all kinds of applications. Also, a large part of the community code base is already written using a message passing framework. For example, all the processes in tightly coupled simulations have to periodically communicate in order to advance the simulation. Therefore, a failure in one of them will affect the others, in a direct or indirect way.

Stencil-based applications. This chapter focuses on a particular application domain decomposition, the Stencil-based layout (which will be described in detail in Section 5.2.1, at the core of many scientific applications that focus on iterative simulations of structured multi-dimensional grids. Examples of Stencil-based applications range from the Jacobi iterative method to partial differential equation (PDE) solvers, including more complex methods such as adaptive mesh refinement (AMR [138]) or multigrid. A number of simulations are considered stencil codes. Examples are S3D, which has been introduced in Chapter 3, or the finite difference discretization of wave equations used in the Reverse Time Migration for seismic imaging [107, 143].

An indicator of the popularity of stencil codes in the community can be found in the great amount of studies focusing on optimizations of certain aspects of Stencil scientific simulations, such as these [43, 132, 109, 153, 94], or the libraries that aim at reusing different stencil optimizations, such as WaLBerla [59], Physis [106], Cactus [71], or LibGeoDecomp [134].

Local Recovery for Stencil-based Applications. This chapter will describe how local recovery can be implemented for stencil applications in an efficient manner and with minimal code transformations. As in the aforementioned "bag of tasks" example, after a failure of a process or node is detected, spare or re-spawned resources can be used to substitute failed ones, and the last checkpoint can be reloaded locally. Since this process does not require the

coordination of all ranks in the domain, recovering locally will be shown to scale perfectly, independently of the total number of ranks in the system. While the recovery process is taking place, the logical neighbor ranks in the stencil domain will probably not have yet noticed the failure. They will do notice it, however, when they begin the synchronization part of each iteration (also known as 'timestep'). At this point, the recovery procedure will indicate the part of the runtime that is running in these ranks where to find the newly recovered logical neighbors.

FenixLR. This chapter will present the requirements, design, implementation, and performance evaluation of the local recovery concept in a new framework, called FenixLR. The main requirements of this framework are the need to share the same interface as Fenix, which API was described in Chapter 4, and to provide efficient, scalable recovery. This chapter will describe how, in order to provide the requirement of efficient recovery, the architecture and implementation of FenixLR required a complete re-design when compared to the constructs in Fenix. FenixLR has been implemented as a standalone runtime whose only dependencies are a C++ compiler and the Cray uGNI library (i.e. no MPI runtime needed). Since it has been built from scratch, this has been a non-trivial task. The motivation behind this decision is that fault tolerant versions of MPI, such as the prototype implementation of ULFM [10, 9, 86, 11, 61] based on OpenMPI, do not serve FenixLR's original requirements. Specifically, they are not capable to deliver local recovery constructs and their global recovery constructs' scalability increases linearly with the number of processes.

Consistency in the Recovery Process. The approach in FenixLR uses implicit coordination as described in Section 4.2.2 to guarantee consistency: a checkpointing mechanism where selective checkpoints are created at specific points within the application, guaranteeing global consistency without requiring a coordination protocol. In order to enable local recovery, FenixLR also logs a small set of messages, which might need to be replayed after recovery. The total size of these logged messages is negligible when comparing with the size of the checkpoints.

Local Recovery Evaluation. FenixLR has been deployed on the Titan Cray-XK7 production system (world's third fastest machine as of November 2016) at ORNL. This chapter presents an experimental evaluation of the effectiveness and scalability of local recovery in

FenixLR using the S3D [33] stencil-based combustion application, which has been presented in Chapter 3.

The results of the evaluation demonstrate FenixLR's ability to tolerate high-frequency dynamically injected node failures while maintaining sustained performance of S3D on scales up to 262144 Titan cores. The 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, it is important to note that the benefits that FenixLR provides for stencil-based applications, compared with Fenix, come with no programming overhead for the user, since FenixLR shares the same interface as Fenix.

Outline. The rest of the chapter is organized as follows. Section 5.2 describes in detail the characteristics and assumptions of Stencil-based codes and the benefits and challenges of local recovery. Section 5.3 presents architectural design of the FenixLR implementation and recovery mechanism. Finally, Section 5.4 experimentally evaluates FenixLR with S3D on Titan.

5.2 Local Recovery for Stencil-based Scientific Applications

This section presents the local recovery approach and the underlying reasoning for exploring it for stencil-based applications. Recovering from failures in a local manner implies that (1) only processes that failed have to rollback to the last checkpoint and (2) only processes that communicate with failed ones will detect 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.

This section first describes the key relevant characteristics of the targeted stencil-based

applications. It then explores the local recovery approach for this class of applications, its benefits in case of single and multiple failures, as well as some of its associated challenges.

5.2.1 Stencil-based Scientific Applications

In this dissertation 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, as shown in Figure 5.1. Each point in the domain can have a set of properties that evolve with time.

A typical scientific application template that falls into this definition can be a simulated 3-D space in which each point has a given set of properties that need to be recalculated over time. To simulate this domain in an scalable way the 3-D space is partitioned in homogeneous subspaces. These subspaces, therefore, can be scattered throughout the processes of the machine.

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 5.2. 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 5.2 is for a 5-point stencil.

In some cases the communication frequency can be decreased by exchanging a bigger ghost region and replicating some of the computation. If data is exchanged in groups of size greater than one, as shown in Figure 5.3, the replicated ghost points have to be computed as if they were internal points. At the end of each iteration, the outmost "*layer*" of the ghost region is discarded. Therefore, in the case of a 2D 5-point stencil, by communicating blocks with thickness n (as opposed to one) the application can reduce the communication frequency: only every n iterations the ghost point buffer has to be exchanged with the logically neighboring ranks.

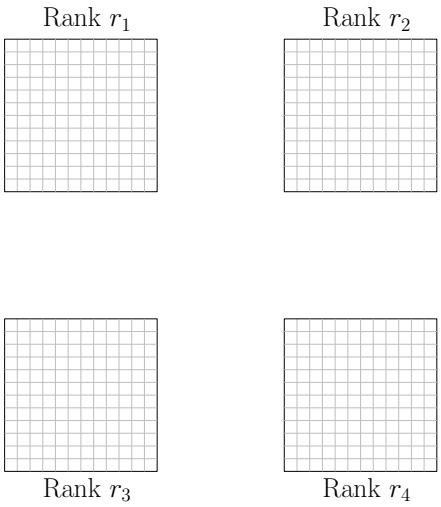


Figure 5.1: Partitioning of a square 2D domain across four MPI ranks.

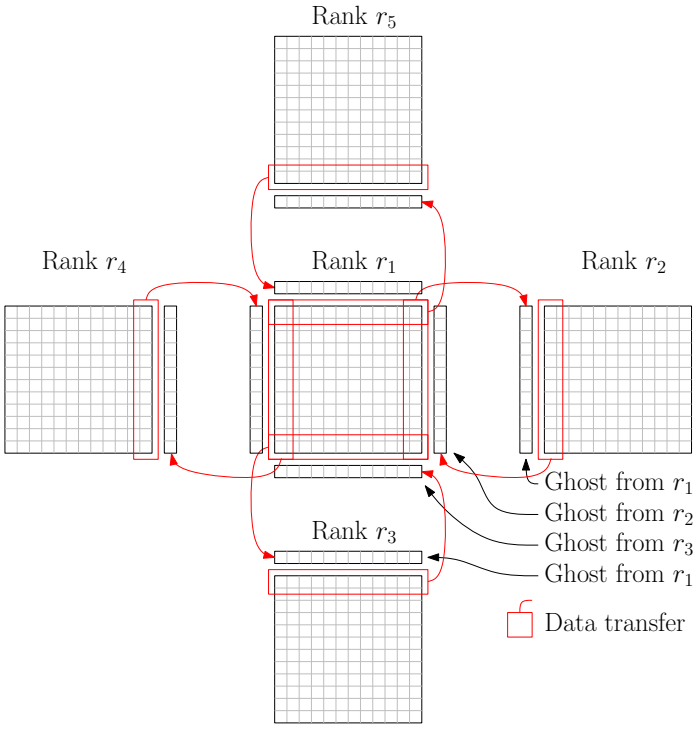


Figure 5.2: Partitioning of a 2D domain across five processes. This figure shows the ghost region buffer exchange between neighboring processes in a typical implementation of a stencil-based parallel application. Note how r_1 maintains a copy of the domain distributed in its neighbors in a special buffer, called the ghost buffer. ©2015 ACM (reprinted with permission) Gamell et al. [67].

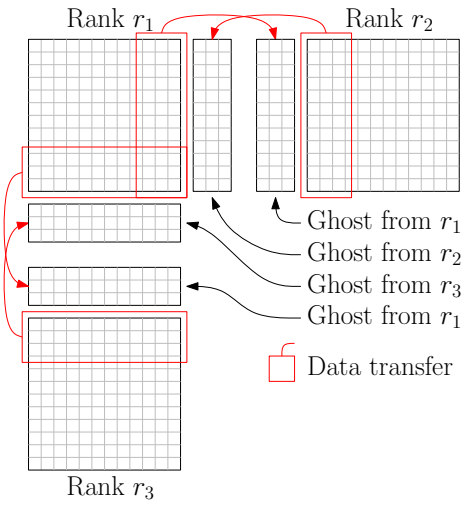


Figure 5.3: This figure shows both the ghost region exchange of two cases: (1) 5-point 2D Stencil in which communication is grouped to reduce its frequency (one communication per three iterations), or (2) 13-point 2D Stencil that communicates every iteration.

Stark et al. study the implementation of a particular stencil mini application using a task decomposition approach [140], which could be extended to include fault tolerance features. In contrast, we focus on a traditional SPMD stencil implementation.

Assumptions about periodic collective synchronization. Not all scientific applications offer the described iterative behavior from the beginning of the execution until the end. Sometimes, collective operations are performed every certain number of timesteps, e.g. for analyzing how the simulation is running, asserting certain properties are still present, or dumping partial results to secondary storage. Even though there is a body of work on how to perform these analysis in-situ, in-transit, and in an opportunistic manner [8, 91], some applications still require a full, machine-wide synchronization primitive, i.e. with an effect similar to a barrier. For example, in case of the S3D application, this interval is typically every 16 minutes, as shown in Chapter 3.

This chapter focuses on the portion of the execution between two subsequent synchronizations, and assume that this interval is long enough so that, in future extreme scale systems, several failures might occur within it. Our goal is to enable this portion of the simulation to run despite the number of failures 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, present since the third

version of the MPI standard, are promising since they can naturally support imbalance between the processes without imposing a barrier-like behavior.

5.2.2 Local Recovery, Challenges and Benefits

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

The observation in the previous section indicates that process failures in stencil-based applications affect the computation of **immediate neighbor subdomains** of lost ones. This allows a natural adaptation of our local recovery approach, leaving the re-spawned process to (1) redo the local computation using the data at the previous timestep restored from the checkpoint, and (2) reestablish communication with the neighbors for the ghost region exchange to proceed the timestepping.

Scalable runtime recovery. After a failure, the environment does not need to be recovered globally, and only the 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 in the job or the system) 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. Despite the relative simplicity of local recovery for target stencil-based applications, the challenge arises to guarantee consistency in the message passing programming model as neighbor processes must communicate to make a progress. Initially, we explored the enforcement of a directional partial synchronization: each process had to send a token to its logical neighbors after successfully transferring the ghost regions and checkpointing the data for that iteration. Once a process received the token from all its respective neighbors and, therefore, had all the data available, it was allowed to continue with the simulation. This guaranteed that, on failure, a process would be able to send again their messages. However, this method was not flexible. The approach we implemented leverages ghost region exchange to implement a domain specific message logging, which keeps the outgoing messages that have been transferred since the last checkpoint. Specifically, for the 1D case, only two messages are stored every timestep: the message sent to the rank logically in the

right, and the message sent to the rank logically located in the left. In 5-point 2D stencil case, for example, four messages, are logged at each timestep. Similarly, in the 7-point 3D case, six 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 since the checkpoint is several orders of magnitude larger. 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 when `MPI_Recv` (or derivates) are re-executed by the processes substituting the failed ones.

The presented approach differs from traditional uncoordinated checkpointing and message logging in several aspects: (1) in the approach presented in this chapter, all created checkpoints are strongly consistent; (2) message logging is used only to enable local recovery, while traditional message logging is used to enable global recovery from a set of non-consistent checkpoints; (3) the message logging in this protocol is sender-based, local, in-memory, and used only by the failed processes; and (4) the presented protocol guarantees that only the failed processes need to rollback.

Note that using protocols such as those presented in [72] with the target applications (i.e., iterative applications with a stencil-based communication pattern), 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 the protocol presented in this chapter, which can guarantee that only the failed process has to rollback while the neighbors can continue.

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).

5.3 FenixLR Implementation

In this section, we first describe our initial attempts at implementing our approach using an MPI-based framework (i.e., ULFM) and the challenges faced. We then present the implementation FenixLR. Note that while FenixLR maintains the same programming interface as Fenix, presented in Section 4.3, it has been built from scratch on top of Cray's uGNI interface – the only component re-used from Fenix is the checkpointing module.

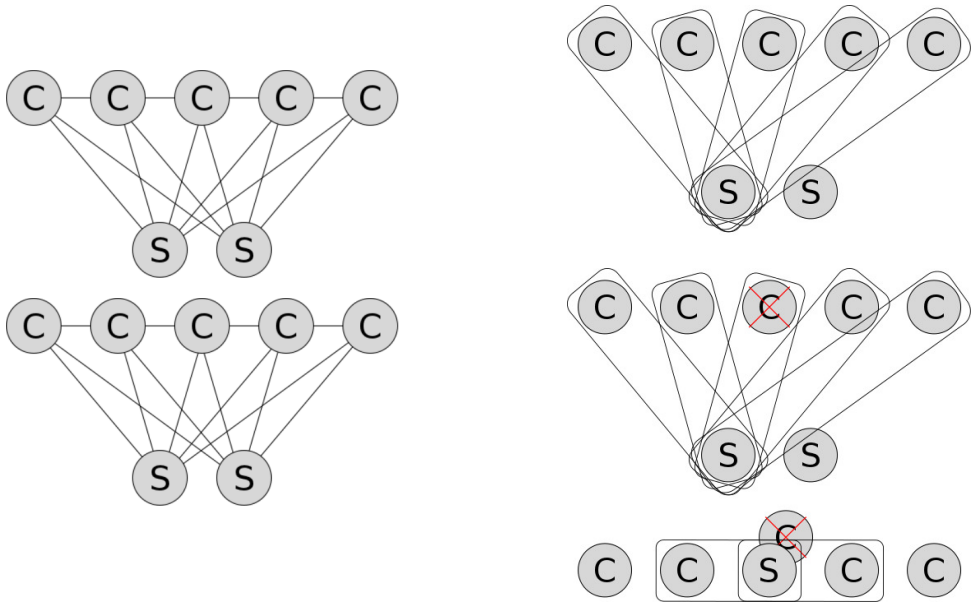
Also note that using local rollback and global continue results in increased complexity as compared to global rollback. When using global rollback, all threads are restarted from a known previous consistent state and all lost computation and communication have to be repeated. As a result, there is no need for the respawned threads to be distinguished from the surviving ones, since all the recovery actions will be the same in all threads.

However, when using local rollback and global continue, the respawned threads may need to receive messages from the non-failed threads that were already sent. As a result, there has to be a mechanism for the respawned threads to request these messages (since the non-failed threads did not rollback).

5.3.1 Experiences with MPI-based Implementations

One of the key operations for communicator recovery is the shrinking communicator operation offered by ULFM, which eliminates failed ranks from the failed communicator, returning a fully functioning communicator with less ranks. The cost of this operation in the ULFM prototype evaluated was not trivial, and we experimentally determined that, as the number of ranks in the communicator increased, the increase in the cost of the shrink implementation was higher than linear. That is why we explored recovery options that avoided the recovery of a global communicator, as described below. No conceptual problems of ULFM prevented its usage but, instead, the limitation was the robustness of the prototype implementation.

Usage of a failed, non-repaired communicator. ULFM specifies that a communicator can be re-used after a failure for point to point communications, without the need to repair it, as long as the failure is acknowledged and no communications are initiated with the



(a) Communicator layout between ranks. (b) Recovery procedure after a failure.

Figure 5.4: Possible implementation of the runtime using MPI but avoiding communicator repair operations. In this version of the implementation, each MPI communicator includes only 2 ranks. A communicator is created between each pair of compute ranks. Each compute rank (C) in a group also requires a binary communicator with each spare rank (S). For scalability, compute ranks are divided in groups and a few spare ranks are assigned to each group. In this example, each group contains five compute ranks and two spare ranks.

failed rank. The first prototype of our local recovery framework leveraged this property of ULFM and, after implementing it and testing it by using a Partial Differential Equation solver on top of the prototype ULFM implementation, inconsistencies were found in the use of some MPI operations in a failed communicator that was not repaired. In most cases, the runtime reached an inconsistent state after a failure and was unable to deliver messages to survivor ranks.

Pair-based communicator layout. After several attempts to fix the aforementioned unpredictable behavior, and provided that ULFM is the only fault tolerance proposal that would allow local recovery, we decided to switch to a different strategy. Since each rank of the targeted application type communicate with a constant number of ranks independent of the total domain size, we designed a layout of communicators composed of only two ranks. The communicators including a rank that failed can be disposed and do not need to be repaired. Assuming no collective operations were required by the application, we designed the creation of a particular communicator for every two process pairs that need to communicate directly. Right before a collective operation, the communicator involving all ranks would need to be repaired.

After a process failure, the processes that were communicating with the failed process are free to dispose of the communicator. In order to allow for a non shrinking recovery mode, the failed processes will be substituted with spare resources. Therefore, for this method to work, the processes neighboring the failed one would be required to have communicators already created with the spare ranks. This results in a number of communicators that does not scale well with the total number of processes in the system and, therefore, we decided to divide the compute ranks into groups of a fixed size, to which a certain number of spare ranks would be associated. All ranks in a group would have a communicator to each of the spare ranks in that group. For a one dimensional stencil case, the communication layout for this possible implementation is exemplified in Figure 5.4a and the recovery mechanism is outlined in Figure 5.4b.

As of version 2 of the MPI standard, in order to set up the communicator structure between spare ranks, we had to create each of them at the beginning of the execution in a collective manner. In order to create the communicators that communicate rank 0 with

rank 1, rank 2 with rank 3, etc. we used `MPI_Comm_split` to divide `MPI_COMM_WORLD` by two, and recursively applying the division process until we obtained communicators with only two ranks. The cost of this operation was $O(log_2(N))$, where N is the total number of ranks in the system. We had to repeat this for the communicators that connect rank 1 with rank 2, rank 3 with rank 4, etc. The total cost of this operation, therefore, is $O(R \cdot log_2(N))$, where R is the number of ranks each rank has to communicate to (i.e. in the 3-point 1D PDE case, $R = 2$).

There were several issues with this solution. For example, communication patterns had to be known beforehand. Also, collective operations would require complete communicator fix, and that would trigger the re-creation of disposable communicators. Another problem is that application code has to be aware of the specific internal recovery mechanisms. Finally, the fact that a subset of all spare ranks are assigned only to one group of compute ranks limits the flexibility of the failure size. For example, if a failure in a group happens to affect a large number of ranks (not an unrealistic assumption, since some failures are correlated), the runtime may not have pre-assigned enough spare ranks to a particular compute group. However, maybe with the total number of spare ranks in the system, the runtime would have been able to recover from that failure.

Note, however, that many of this limitations could be overcome if communicators could be created in a non-global manner without the need of point to point communication over a failed communicator. This could be achieved if only the ranks participating in the communicator were needed in order to create the communicator. Methodologies such as [46] cannot be applied in our situation because `MPI_Intercomm_create` uses point to point communication between ranks in a failed communicator, which, as explained above, is unreliable upon failure. By using MPI 3 `MPI_Comm_create_group` operation (assuming it is implemented in a different manner than the described in the above reference), non-collective communication creation would allow a cost of O(R) to create the topology described in Figure 5.4a. Also, all the links between the compute ranks and the spare ranks could be eliminated, as this can be created upon failure.

This second prototype was also implemented and tested with a one-dimensional PDE solver. Even though the prototype showed demonstrated that the concept could work,

the aforementioned disadvantages of this method were apparent and, therefore, made this solution infeasible for other, more complex, stencil computations.

Therefore, we decided to implement a more robust prototype for the FenixLR runtime directly using uGNI constructs, avoiding the use of the MPI runtime altogether. We understand that MPI-based frameworks in general and ULFM specifically are evolving rapidly and we will revisit them once the above issues are fixed.

5.3.2 Implementation Overview

FenixLR architecture is layered and modular. Four key modules collaborate to achieve its full functionality. First, the module dedicated to *communication* offers a base class *Command* that abstracts out the details for a reliable request of service or a reliable communication with other ranks using a particular transport layer. This is used to abstract out the fault tolerance mechanisms from the particular operations (such as the send or the receive of a message, a collective barrier, broadcast, or reduction, or even the transfer of checkpoint data), which only extend the base class and are only required to implement the logic of each operation. Second, the module dedicated to *process resiliency* implements the protocols used to locally repairing the environment after a failure, which includes but is not limited to the management of the pool of spare processes. Third, the module dedicated to *data resiliency* implements the methods for creating, storing, and recovering checkpoints. In particular, it implements a neighbor-based checkpointing such as the one described in Chapter 4. This module offers a well-defined interface so that libraries optimized for other data resilience methods can be plugged in instead. Finally, the *transport layer* also offers a well defined interface to ease the portability to different physical communication APIs. In our implementation, this interface has been implemented on top of uGNI, the Cray transport layer API. In the event of a failure, the runtime takes care of orchestrating the necessary modules during the recovery process as well as determining operations required to be re-executed.

FenixLR allows the dynamic connection between ranks, which is key toward recovery. This connection is implemented through a handshake process that logically connects both ranks. When a process fails, a spare process is used in its place. This spare process

re-creates the handshake process with all processes that were logically connected with the failed process. While the handshake to recreate connections is running, the failure is notified to the rest of the domain through a background collective operation so that new process that try to connect to the failed rank redirect the handshake request to the corresponding spare process. If a new process tries to connect to the failed process before receiving the collective notification, the failure will be detected and that process will start the failure recovery procedure by contacting an available spare rank. However, this will notify that the failure has been already recovered and will point to the correct spare rank that is substituting the failed process. The same behavior is reproduced for multi-process failures, which are considered simply a set of process failures.

FenixLR offers three language bindings: C, C++ and Fortran. We maintained the same programming interface as Fenix (Chapter 4), as its low programming overhead has been demonstrated – it required less than 35 new, changed, or rearranged lines of code for S3D.

In fact programming local recovery using FenixLR is even simpler than programming 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 – survivor status is no longer valid, since all survivor processes do not get interrupted from their processing. This simplifies the application side of the recovery logic.

5.4 Experimental Evaluation

This section presents an experimental evaluation of the effectiveness and performance of the local recovery techniques implemented in FenixLR using the S3D combustion simulation on the Titan Cray XK7 at ORNL.

Chapter 4 shows that Fenix can tolerate failures in an online and global manner with failures occurring every 47 seconds. The implementation used in Chapter 4 leveraged MPI-ULFM to detect failures and recover from them. In the implementation presented in the current chapter, the sources of overheads have been reduced, and the fault tolerance mechanisms have been implemented on top of uGNI, Cray's interconnect API. This section experimentally evaluates how these algorithmic improvements can enable FenixLR to effectively handle even higher failure rates (e.g. unrelated node failures coming every 5 seconds).

5.4.1 Goal

In summary, the goals of the experimental evaluation presented in this section are to demonstrate that (i) using the local recovery technique presented in this chapter, FenixLR can enable tightly-coupled stencil-based applications, such as S3D, to recover from node failures occurring as frequently as every 5 seconds, (ii) failure recovery is scalable, and (iii) failure recovery overhead does not need to be proportional to the system size when recovering locally.

5.4.2 Methodology

In what follows, the experimentation methodology is presented and the experiments are described in detail.

This section first presents an evaluation of the asynchronous checkpoint transfer technique and compare it with traditional synchronous technique, both implemented on FenixLR. This evaluation includes an scalability test on both data size and total number of processes and an study of asynchronous data transfer impact on S3D iteration time. Checkpoint size has been forced to 130 MB/core, even though current S3D production runs checkpoint in the order of 5 MB/core.

This section then studies the overheads related to the recovery process and its scalability. In this experiment, *worst-case* failures are injected, i.e., sets of failures that do not allow recovery propagation delays to merge (an effect that would produce several failures to mask each other as described in Chapter 6) and, therefore, the total overhead is the sum of the recovery overhead for each failure. It is shown that FenixLR handles extreme node failures (e.g., MTBFs as low as 5 seconds) with total overheads of up to 50% in the worst case scenario. These results empirically show that this method can be more efficient than theoretical full redundancy (which, assuming constant resources would have a temporal overhead of at least 100%), for the targeted class of applications.

A key goal of this evaluation is to study how the presented approach behaves at current scales, and use this to try to explore behaviors and performance at exascale. As a result, the experiments were conducted on up to 262,272 cores. In order to perform these experiments node failures are injected by simultaneously sending SIGKILL signals to all application

processes running on a particular node. As the network setup parameters are stored in process memory, when killing all processes in a node, no software disconnections are allowed with connected ranks – this is consistent with the behavior of a real node failure. The processes on the other nodes receive error codes when trying to perform a uGNI operation with any of the processes that were killed. In this sense, the injected failures can be considered *real* failures, as opposed to pretending that a process has failed by, for example, setting a flag in-memory. In what follows, 'failures' refers to '*node* failures', which are equivalent to *N-process* failures, where N is the total number of processes on a system node, blade, etc. By default, the experiments use $N = 16$. Unless specified otherwise, all tests have been repeated 5 times. Error bars in all Figures show the average, first quartile, third quartile, maximum and minimum of the 5 repetitions.

Only the experiments with MTBFs lower than a minute are displayed due to the fact that higher MTBFs (such as 5 minutes, 10 minutes, 30 minutes, etc.) show negligible recovery overheads relative to the total execution time.

All the experiments were performed on the Cray XK7 Titan at ORNL. Titan is composed of 18688 16-core CPUs 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.

5.4.3 Asynchronous Checkpoint Transfer Cost and Scalability

The goal of the first experiment is to evaluate in-memory asynchronous-transfer checkpoint performance and scalability. Chapter 4 evaluates double in-memory checkpointing and demonstrates that it scales independently of the number of processes.

We first reevaluate the scalability of the checkpointing mechanism implemented in FenixLR in the event of failures, as well as with per-core data sizes sixteen times larger than the ones evaluated in Chapter 4. Figure 5.5 represents the study of weak scalability up to 262272 cores and checkpoint sizes of 130 MB per core while injecting failures every 10 seconds. It can be concluded that checkpointing overhead is constant and independent of

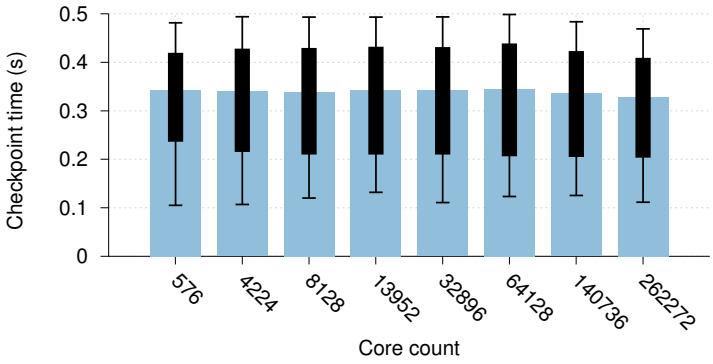


Figure 5.5: Weak scalability of FenixLR's asynchronous checkpointing. Checkpoint size is set to 130MB/core in all cases. As shown, job size increases do not impact checkpointing performance, which only depends on the per-core checkpoint size rather than the total per-job checkpoint size. ©2015 ACM (reprinted with permission) Gamell et al. [68].

the total core count even in the event of failures, which proves as perfect weak scalability.

Impact on iteration time. Our checkpoint methodology consists of creating a local copy of the data (e.g. `memcpy()`) and immediately return the control back to the application. In the background, the data is sent to its destination while the processor is computing the next iteration. Figure 5.6 shows the impact of checkpoint transfer on total iteration time for different data sizes. For each data size shown in the X-axis we can observe asynchronous checkpoint overhead in blue and synchronous checkpoint in green. In both bars, light color refers to the iteration time itself, and bold color represents the time inside the checkpointing call itself. As expected, the iteration time suffers from the overhead of the transfer. However, the total iteration time (including the checkpoint creation and storage) is, in all cases, reduced by using asynchronous checkpointing.

In this dissertation we did not tune performance by understanding S3D communication patterns, so Figure 5.6 represents a worst-case scenario. The runtime system has this information available and could potentially analyze it in a transparent manner, which would dramatically decrease the impact of checkpoint transfer on the iteration time. We leave this improvement as future work.

5.4.4 Recovery Time for different MTBFs

The goal of this second experiment is to demonstrate that FenixLR is capable of handling high-frequency failures occurring up to every 5 seconds. This experiment also studies the

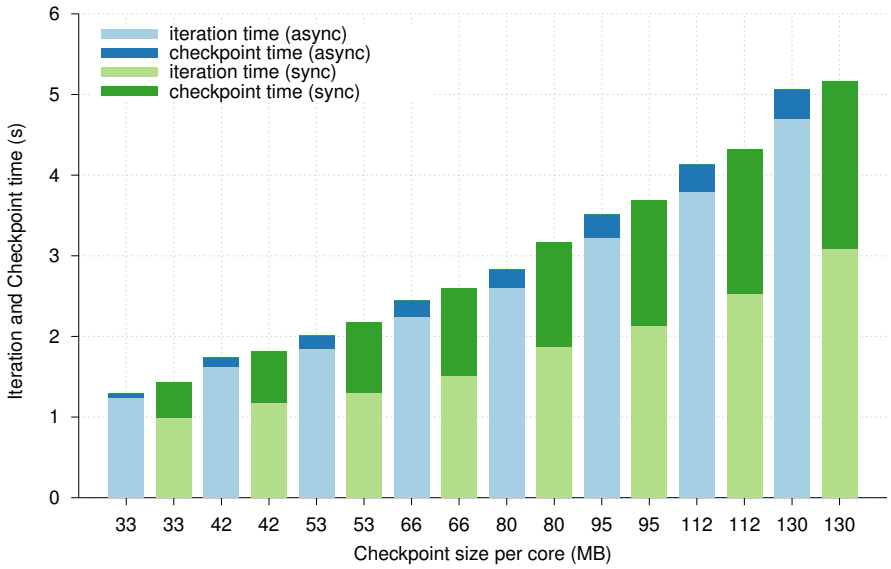
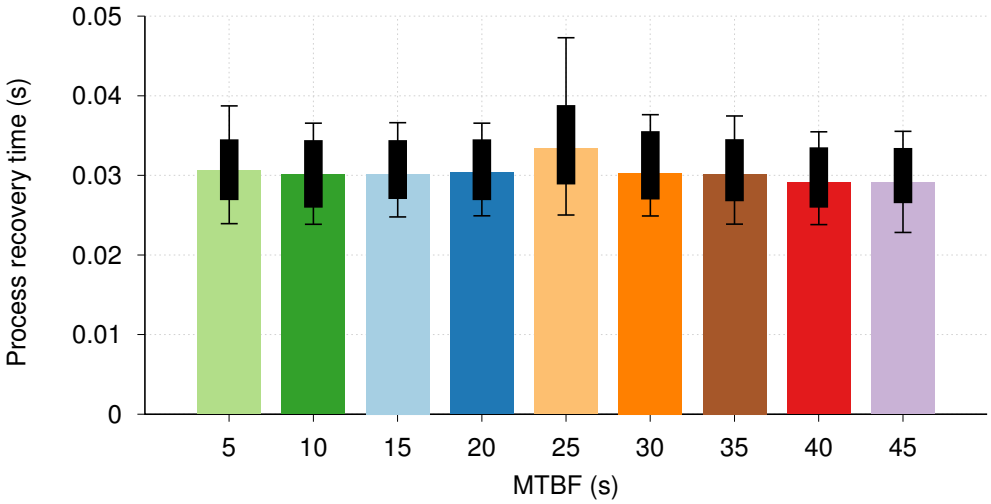


Figure 5.6: Comparison of total iteration time (including checkpoint) using synchronous and asynchronous checkpointing, for different problem sizes using a total of 4096 cores. This figure shows that overlapping communication and computation is possible and beneficial in S3D. This experiment was performed without injecting failures.

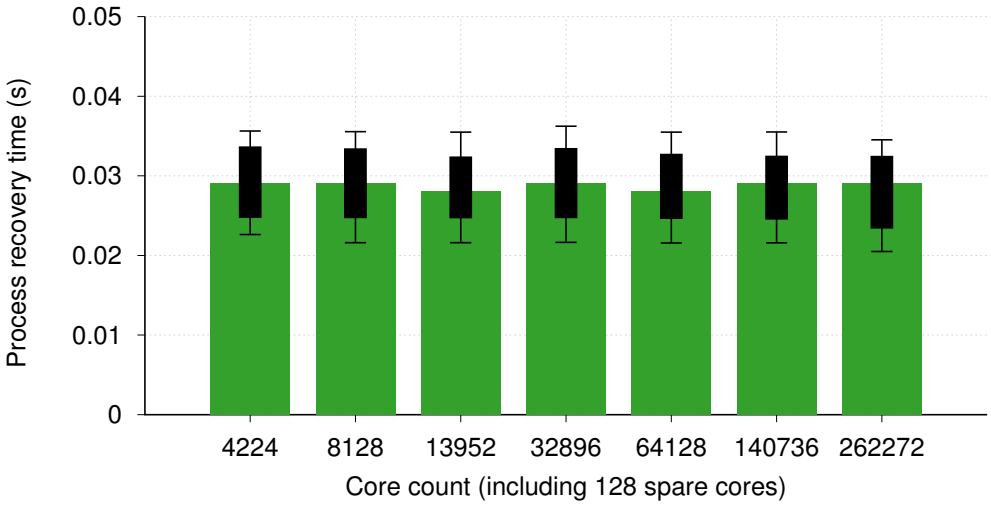
overheads due to process recovery and empirically demonstrates that the performance of FenixLR does not depend on the total number of processes (i.e. demonstrates good scalability), which is highly desirable at exascale. These experiments were performed using S3D on 4736 cores (4096 compute and 640 spare cores), unless otherwise specified.

This experiment also aims to explore the benefits of local recovery when compared to global recovery. Consequently, in this experiment the set of failures injected are engineered so that they do not allow propagation delays due to local recovery to merge, and therefore the total overhead is the sum of the recovery overhead for each failure. In other words, this experiment evaluates the worst-case local recovery overhead without the failure masking effect, which is described and evaluated in Chapter 6.

Process recovery overhead. First, the total overhead of recovering from a failure is studied for different failure frequencies. Figure 5.7a plots the average overhead of process recovery for different frequencies of node failure injected during an S3D execution of about 200 seconds. Each bar represents the average time to recover the processes from a failure. This overhead is initially seen only in the spare processes that substitute the failed ones, and then propagates to the rest of the domain due to the communication pattern of the



(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 5.7: Time to recover from process failures with varying frequency of failure arrival as well as varying total number of cores in the job [68]. Note that the recovery process is perfectly scalable, tested up to 250+ thousand cores. ©2015 ACM (reprinted with permission) Gamell et al. [68].

application. This figure does not include overheads due to data recovery (i.e., fetching the checkpoint). Note that fetching the checkpoint is the exact inverse process of checkpointing: First, the checkpoint has to be fetched from the neighbor that stores it in-memory to the memory of the spare process. Then, it has to be copied using `memcpy()` to the application memory. Therefore, the overhead of fetching the checkpoint will be similar to the checkpointing overhead, shown in Figure 5.5.

Scalability. Figure 5.7b plots the average process recovery time for every failure. The figure demonstrates that, for experiments on up to 262272 cores, the local recovery overhead is constant and independent of the number of processes in the system. These experiments were performed with a fixed MTBF of 10 seconds, and all the experiments used 128 spare processes. Furthermore, experiments for system sizes smaller than 64k have been repeated with a different total number of node failures, ranging from 1 to 8 node failures, and the results have been averaged in the plots.

5.4.5 Total Overhead for different MTBFs

Failure Frequency. The previous experiments demonstrated that the process recovery time for a single failure is small, and is constant and independent of the total number of cores used by the application as well as the frequency of failures. The next set of experiments study the total overhead due to fault tolerance, i.e., including overheads due to checkpointing, process/data recovery and rollback. The goal of these experiments is to compare the end-to-end execution time of a failure-free, checkpoint-free execution with the end-to-end execution time with different MTBFs. The experiment is run using a fixed core count of 4096 cores and 640 spare processes and a checkpoint size of 53 MB/core.

Figure 5.8 plots the results of the experiment. For different failure rates ranging from 5 to 45 seconds, Figure 5.8 plots the total overhead relative to a failure-free, checkpoint-free execution base case. The total number of failures ranges from 48 processes (3 nodes) to 528 processes (33 nodes), as noted on top of each bar, during a total time of about 150 seconds.

The right-most bar in the figure shows the overhead of the same experiment but using global recovery while injecting failures every 47 seconds, as described in Chapter 4. It can be seen that by using local recovery the performance penalties are much lower, even at higher

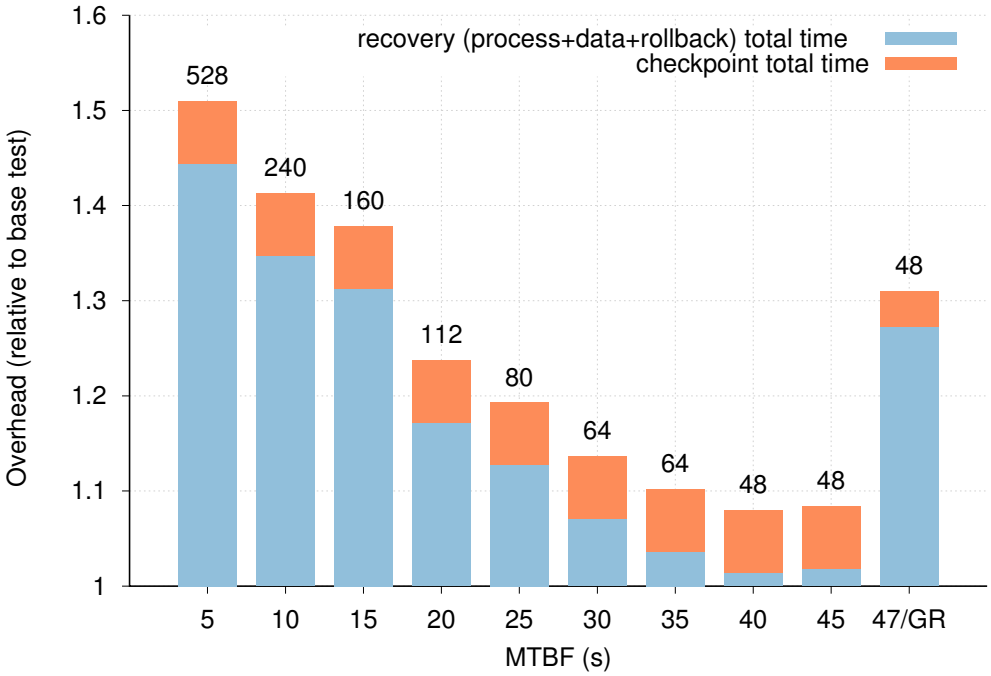


Figure 5.8: Overall failure overhead of different MTBFs relative to a checkpoint-free, failure-free base test execution. On top of each bar the total number of process failures recovered throughout the execution is indicated. Job size has been fixed to 4736 cores, corresponding to 4096 compute cores (an S3D domain decomposed in a grid of 16^3 cores) and 640 spare cores, and each checkpoint requires 217 GB. ©2015 ACM (reprinted with permission) Gamell et al. [68].

failure rates. For example, note 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). Also note that the total overhead of running the experiment in the worst-case scenario (i.e., with node failures every 5 seconds), is 51%. That is, it is slightly worse (i.e., 1% worse) than the theoretical best-case overhead of using 2-way redundancy. Again, it is important to emphasize that in this experiment, we injected worst-case failures, i.e., in which failure masking (an effect that will be described in Chapter 6) does *not* occur in most cases – the only exception is the 5-s MTBF experiment, in which the total time is just a small portion above the 10-s MTBF test. This is intentional, and has been done to study only one of the direct advantages of process recovery as well as its scalability. Chapter 6 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 5.8 by the number of node failures.

5.4.6 Evaluation Conclusion

These experiments demonstrate how the local recovery algorithm can tolerate multiple unrelated failures that strike as frequently as every 5 seconds with lower overheads than global recovery.

Chapter 6

Failure Masking on Stencil-based Applications

This chapter aims at experimentally and analytically studying failure masking, which can occur when multiple failures affecting the same execution of a stencil application are recovered locally: the failures may mask each other. This dissertation considers N failures to mask each other when the effect of such failures in the end-to-end time is proportional to k, being k smaller than N. For example, two failures can mask each other if their combined effect on the total execution time is the same as if only one of the two occurred.

6.1 Overview

Local Recovery in Stencil-based Applications. Chapter 5 looked at how to recover from failures in a local manner. In codes that are organized following a stencil pattern, as described in 5.2.1, when a failure strikes, spare resources can substitute for any affected processes or nodes. These spare resources can then reload and rollback from a previously stored checkpoint. This process is done without the interruption of the logical neighbors. They will, however, notice the failure when they begin the synchronization part of each iteration or timestep. These immediate logical neighbors will therefore need to stall at this point, waiting for the respawned ranks to catch up with the simulation.

Delay Propagation. Notice, however, that this layer of immediate neighbors L_1 *will* be able to synchronize with their own set of neighbors, which are in a second layer L_2 (e.g. its distance from the failed process/node is two "hops" in the simulated domain). L_2 cores will stall, however, on the *next* iteration. This is a recursive process that will eventually reach all the machine, and can be seen as a "wave" that originates in the failure point and propagates through the processes of the machine.

Failure Masking. Essentially, if iterations are assumed to be long enough (e.g. 5 seconds), the propagation time will also be long. By using this principle, it can be seen how a second failure could occur in a distant node *before* the original failure has spread to that node. Of course, the recovery on this second failure will create another "wave" that will begin to propagate. At some point in space and time, these two waves will merge, but their effects will not be combined in an additive manner. Only the maximum of both overheads will continue to propagate. In short, this means that the overhead on the total execution time of two separate failures in space and time can appear to be as the recovery overhead of a single failure. This can also be true with more than two failures, as it is shown in the experimental evaluation. Note that the larger the machine is, the more plausible this effect becomes. It is for this reason that the described scenario would be optimal for future extreme-scale HPC systems.

Outline. The rest of the chapter is organized as follows. Section 6.2 describes in detail the delay propagation characteristics and the theoretical benefits of failure masking. Section 6.3 presents an application-centric model of the delay propagation effect and, based on it, Section 6.4 simulates the arrival of several failures to provide insights on failure masking. Section 6.5 and Section 6.6 study two different methods to increase the probability of failure masking through decreasing the communication frequency among the different processes. Section 6.7 experimentally injects real node failures in S3D while running on top of FenixLR on Titan to show how the failure masking effect really occurs.

This chapter contains portions adapted from published papers by Gamell et al. [68, 67, 69] (adapted with permission) ©ACM 2015, ©SIAM 2017.

6.2 Impact of Recovery Delay Propagation on Failure Masking

6.2.1 Delay Propagation

Using local recovery, when a failure occurs, neighboring processes can still move forward with their computations during recovery, while the spare process is being notified and the recovery process begins. However, the neighboring processes will eventually need data from the restored process in order to continue, and this data will not be available until the

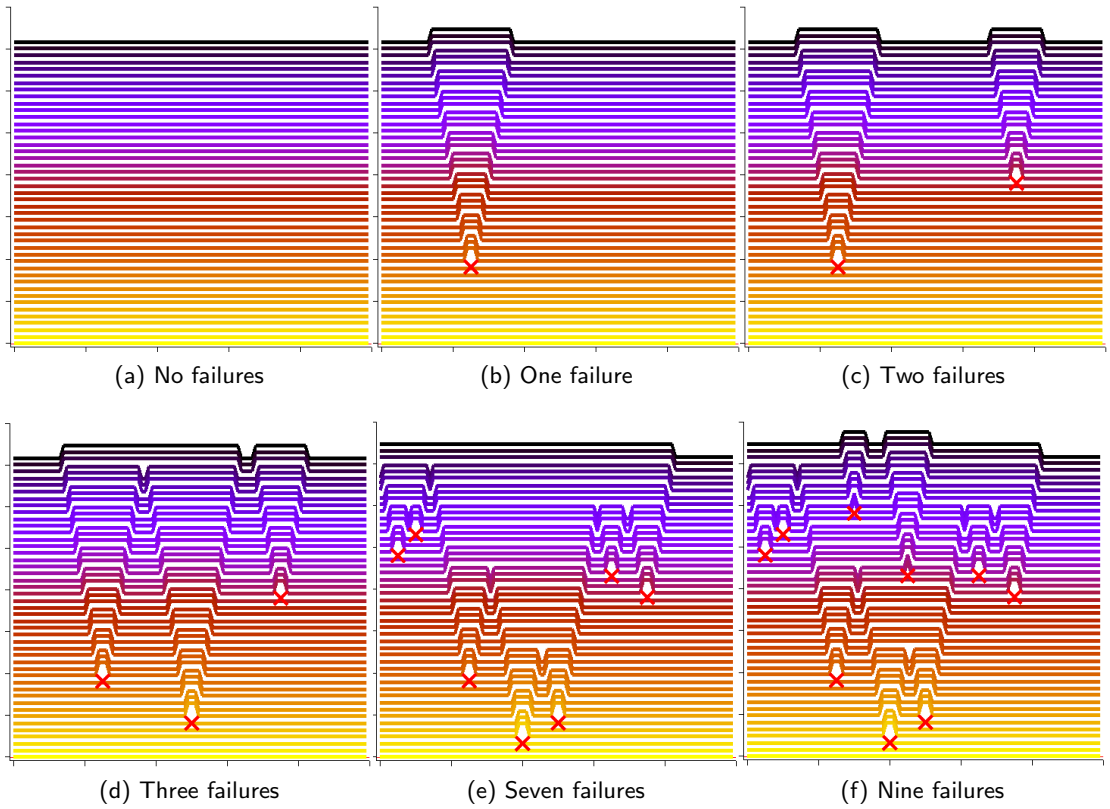


Figure 6.1: Behavior of local recovery for a stencil-based 1-D partial differential equation (PDE) solvers. X axis represents process number (or rank) and Y axis indicates wallclock time. Each line in a figure represents a timestep, and the color of the line represents how advanced the simulation is (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. Instead, 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 6.1b, 6.1c, 6.1d and 6.1e 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 6.1f, however, the total recovery time is equal to sequentially recovering from two failures. ©2015 ACM (reprinted with permission) Gamell et al. [67].

restored process has fully recovered (i.e., recomputed all lost work) and progressed to the next communication. In other words, 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 6.1b for a 1-D stencil. While we use 1-D to illustrate the process, this behavior also applies to higher dimensions.

6.2.2 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 seen in Figure 6.1c. 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, as opposed to the addition

of the two. We call this effect *failure masking*, and an example can be seen in Figure 6.1d. 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 6.1d and 6.1e. Comparing these four figures, we see that the total overhead is the same. This effect becomes more plausible with larger machines and lower MTBFs, 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 6.1f, in which the total execution time is comparable to that of recoverying 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 on (1) the dimension of the application domain (i.e. 1D, 2D, 3D), (2) 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).

Better-than-linear scalability of local recovery. To sum up, local recovery has ideal scalability in terms of resilience overhead for Stencil-like applications due to failure masking: (1) as the machine size grow, failures are more likely to happen with lower MTBFs and (2) unrelated failures are distributed homogeneously among processes.

Further containing the effect of failures. One potential optimization of our approach is the containment of the delay through the overlap of communication and computation. For example, a process can finish all of its local computations that do not depend on the incoming message from the slower neighbors, and can even initiate the next round of communication of the ghost region with the un-affected neighbors so that these do not have to be delayed yet. This communication can be initiated provided the computation of the domain's borders were able to finish. In other words, the delay is masked by the local computation, diminishing the length of the outstanding delay.

The following sections study this 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.

6.3 Modeling Delay Propagation

Stencil-based computations and how they can benefit from local recovery are described in Chapter 5, while failure masking is detailed in Section 6.2. 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 patterns for multi-dimensional stencil-based computations experiencing failures that may mask each other due to local recovery.

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 [42]. Our models differ from those presented by Datta in that we focus on inter-node performance and study how the imbalance due to factors such as failures affect the execution.

Section 5.2 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:

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

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.

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.

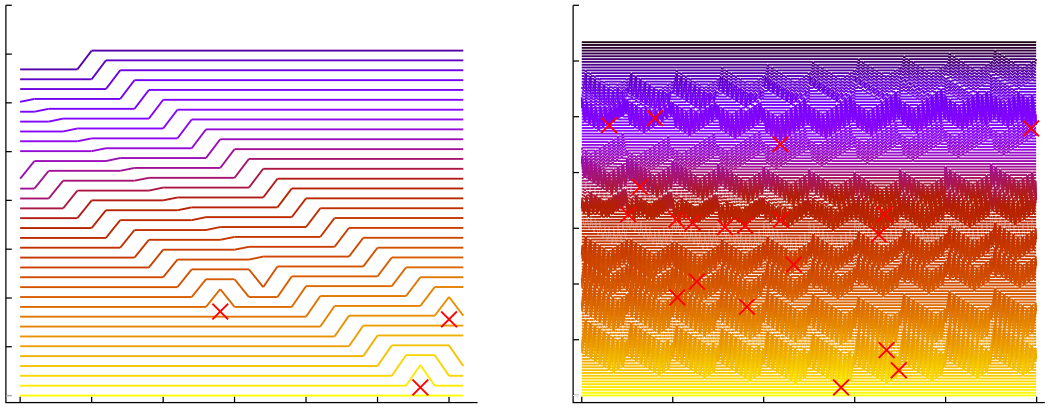
As empirical results presented in Section 6.7 corroborate, the results and assumptions in this model accurately represent the behavior of both simple 1-D PDEs as well as complex stencil-based codes such as the S3D application.

6.4 Failure Masking Analysis

Based on the concepts within the model presented in Section 6.3 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. Even though our experimental evaluation presented in Section 6.7 emulates failures using MTBF, 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



(a) 1D simulation, 32 iterations

(b) 3D simulation, 100 iterations

Figure 6.2: Execution pattern obtained through a simulation based on the presented model. Figure 6.2a represents a uni-dimensional stencil and uses 32 processing elements, while Figure 6.2b represents a three-dimensional stencil and uses one thousand processing elements. Figures axes have the same meaning as in Figure 6.1: 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.

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.

6.4.1 Propagation of a Multi-failure Recovery Delay on 1-D and 3-D Simulations

The simulations reproduce the progress of each timestep as presented in Figure 6.2. For example, on Figure 6.2a 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 Figures 6.13 and 6.14 presented in Section 6.7). Another example can be seen in Figure 6.2b, in which a 3-D 7-point stencil on a $100 \times 100 \times 100$ domain is simulated while twenty failures are injected. The failure propagation in the 3-D case is not as straightforward to distinguish as in the 1-D case, since the X axis of the plot indicates the processing element number, and the mapping from the processing element number to a

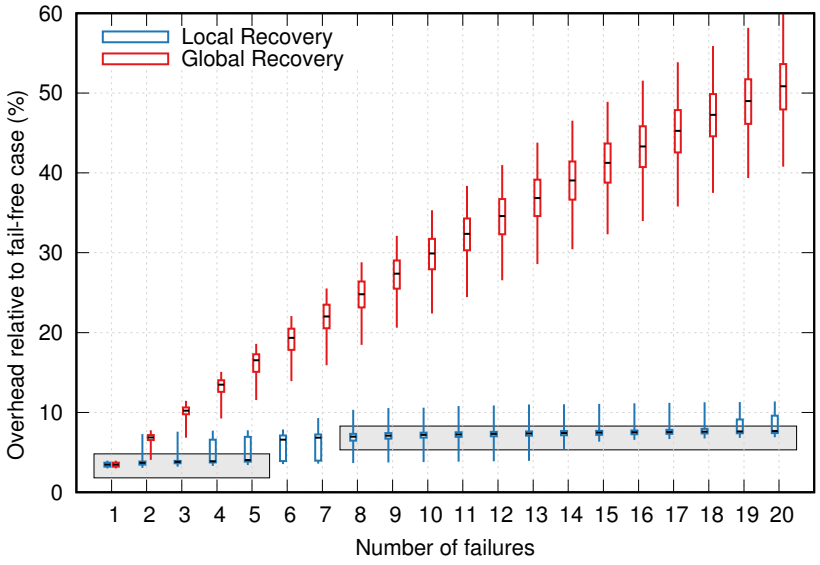


Figure 6.3: Recovery overheads for local and global recovery obtained from simulations based on the presented model for the parallel 3D stencil code running on $100 \times 100 \times 100$ processing elements. In this plot, each candlestick represents the minimum, maximum, median, first and third quartiles overhead of 10048 simulations. Each simulation runs 100 iterations with $T_{it} = 100$ and $T_R = 300$.

3-D position is not visually trivial. However, as in the 1D case, very similar results can be observed between the modeled 3-D case and the actual executions presented in Section 6.7.

6.4.2 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 executed the simulation for 10048 times. In every sample, we generated a new timestamp and processing element number for each failure injected. We then plotted the minimum, first quartile, median, third quartile, and maximum overhead on the execution time for local recovery and global recovery in Figure 6.3.

In case of the global recovery model, we applied the same delay factor $T_R + T_{rollback}$ to all the processing elements when a failure occurs in one (or multiple) 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 Chapter 4. We do, however, apply the same delay effect to both cases in order to observe

the benefits obtained only through failure masking.

The model-based simulation shows that 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 6.3 by two horizontal rectangles (shown as gray in the 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, in which their third quartiles approach the 2-failure global recovery mark ($\sim$7.5%). 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 6.3 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.

6.4.3 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, a process or node may fail after it has experienced the delay due to the prior failure of a process or node. In this case, the delays due to the local recovery of the two failures will add rather than mask each other.

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 multiple failures masking as a single failure, and generalize it to understand the failure density thresholds $p_{be,n}$ for masking failures as if n failures occurred, for $n \geq 1$. 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_{it} is the number of iterations, and N_C is the number of iterations between communication. We also assume that N_T is the number of processing elements and 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).

Layered Propagation Delay. It can be inferred from the model in Section 6.3 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 6.4 and Figure 6.5, which will be described and analyzed in Section 6.4.4.

6.4.4 Failure Overhead Distribution for Multi-failure Global and Local Recovery

Figure 6.4 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 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 ∼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[1]. The first two tests were evaluated with a recovery time of $T_R = 300$ time units to simulate a production-level scenario (see Chapter 3); in the third test the recovery 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

[1]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 ∼51 times larger than one that fits in Titan.

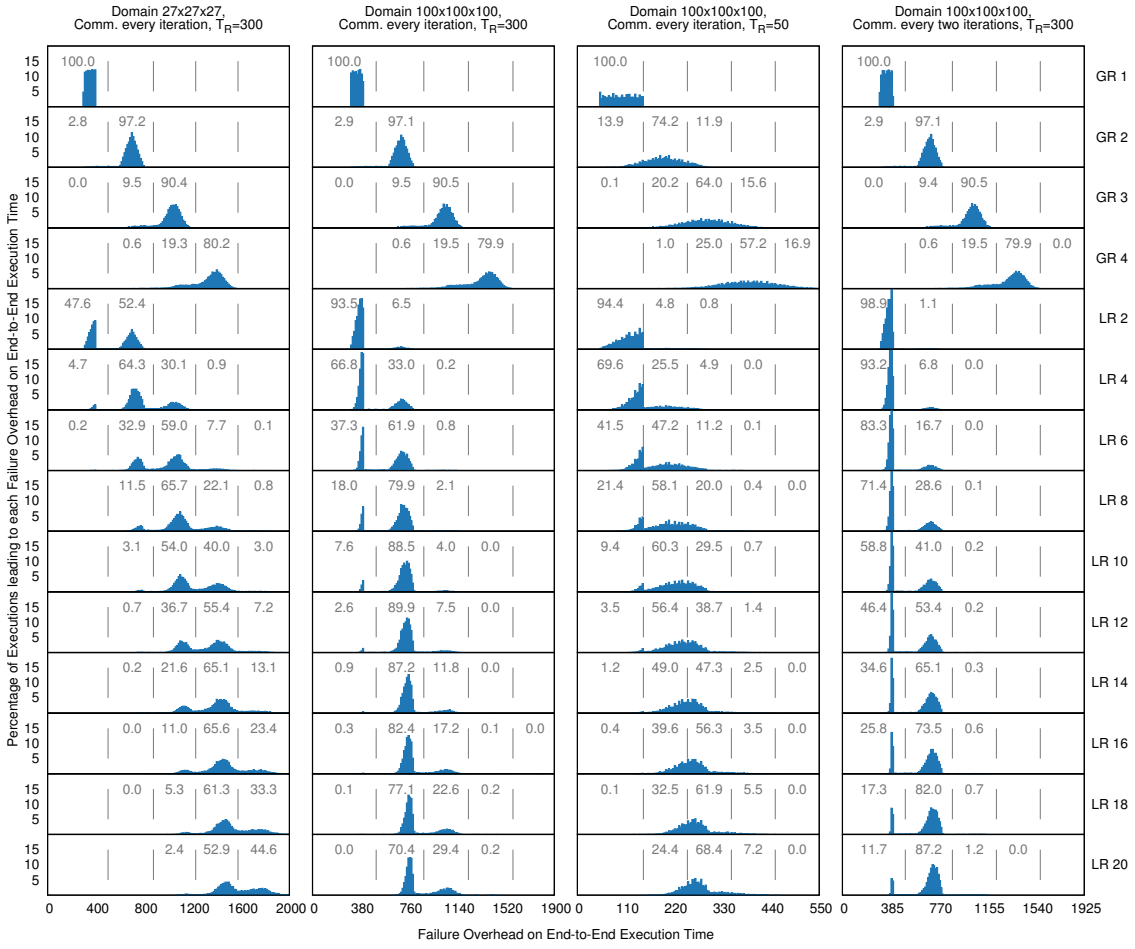


Figure 6.4: Histogram of failure overheads of four configurations. For each configuration, fourteen histograms are shown: four with Global Recovery (GR1 to GR4) and ten with Local Recovery (LR2 to LR20). The number i in GRi or LRi indicates the number of failures injected in each test shown in a particular 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 each two lines, which are 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).

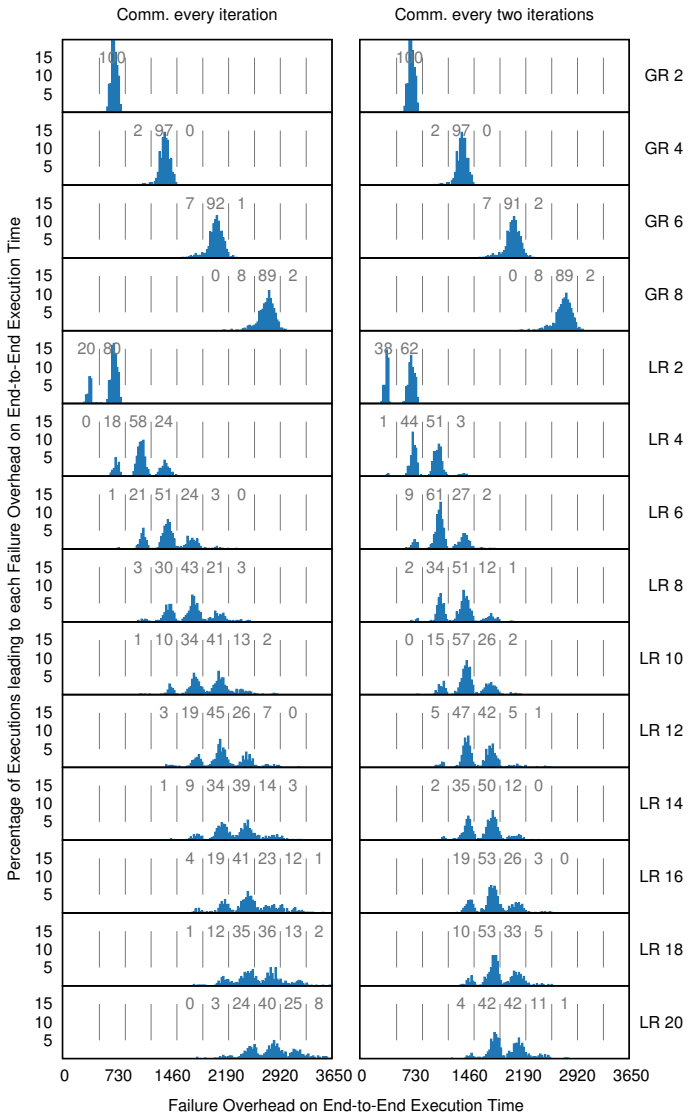


Figure 6.5: Histogram of failure overheads of two configurations. In both cases, experiments simulate 1000 iterations of a domain decomposed into $100 \times 100 \times 100$ processing elements and use $T_R = 300$. Otherwise, the configurations and format are identical to those in Figure 6.4.

two, four, six, ..., twenty failures injected). Each of these histograms represent the overhead imposed by the respective failure recovery mechanism across 10048 samples[2]. The only variation per sample was the failure position in space and time (i.e., the timestamp of every failure and the affected processing element) 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 number '100' (in gray) indicates that 100% of the samples (i.e., all 10048 samples) fall before the first vertical 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 the 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 6.4 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

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

(second case in Figure 6.4). 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 6.4), 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 (∼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 6.4 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 ∼60% of the samples 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 6.5 explores two extra configurations by injecting failures at a much smaller density. In particular, we inject the same number of failures as in Figure 6.4 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.

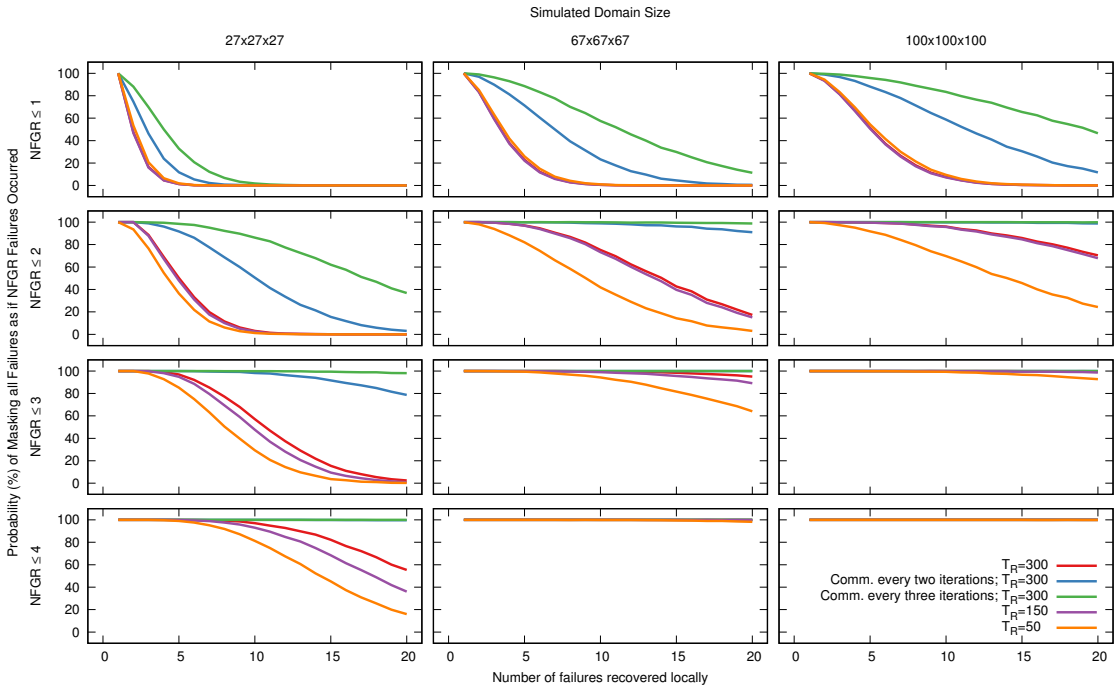


Figure 6.6: Probability of masking multiple failures as a single failure (top row of plots), as two or less failures (second row of plots), as three or less failures (third row of plots), or as four or less failures (last row of plots). The Y-axis indicates the probability, from 0 to 100%, that a particular 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.

Conclusion. Figure 6.4 and Figure 6.5 decomposes the simulation samples in a full histogram for a more in-depth analysis than the error-bar representation shown in Figure 6.3. It also goes one step further by studying the impact of different application behaviors (e.g., domain and machine size, iteration length, communication duration and frequency, or different recovery times) on the probability and level of failure masking. Figure 6.4 and Figure 6.5 reaffirms the conclusions drawn in Section 6.4.2, 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.

6.4.5 Failure Masking Probability

Figure 6.6 plots the cumulative probability of a certain number of failures (X axis), 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, or 4} failures. The left column of the 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 impact other cases: shorter recovery times (which are desirable in order to minimize the overhead in end-to-end time) decrease the probability of failure masking when high failure counts occur.

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 order to maximize the probability 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. [75], 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.

6.4.6 Impact of Performance Variation

As explained in Section 6.3, performance variation often impact 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.

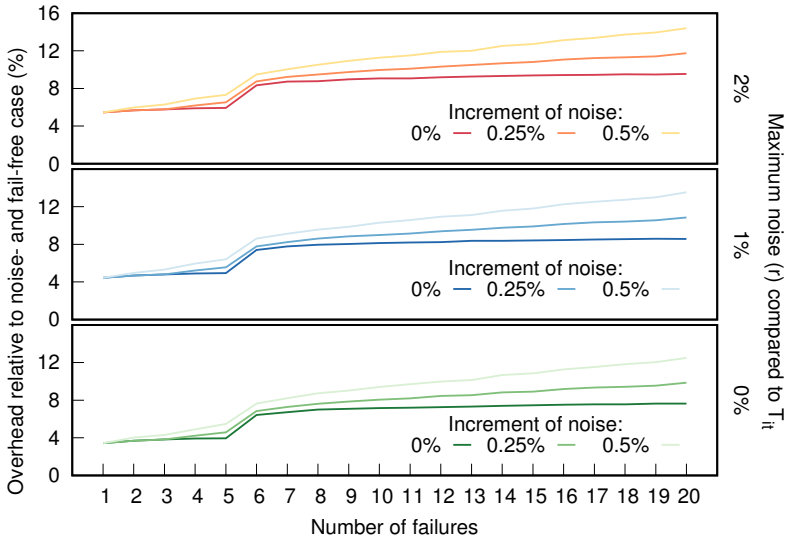
Figure 6.7a shows how an increasing number of failures is masked for different values of r and while increasing r after each failure. The sudden increase when moving from five to six failures is the same effect explained when describing Figure 6.3. Results show that noise does not affect the probability 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 6.3) we can see that failure overheads are significantly reduced due to masking.

Figure 6.7b 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 6.7b 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}.

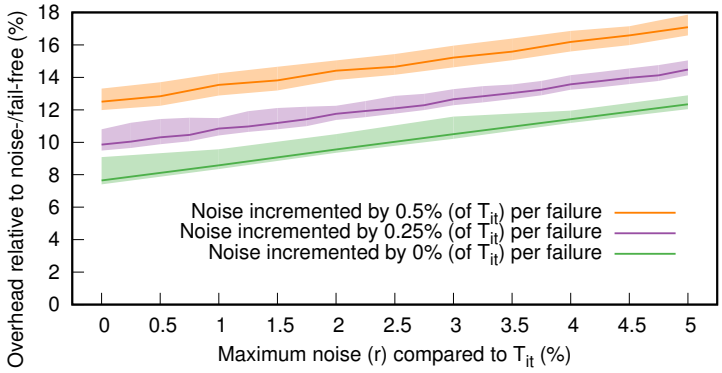
6.4.7 Defining Time Units and Processing Elements

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 section 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



(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.

Figure 6.7: 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 6.3 and solid lines represent the median of 10048 repetitions.

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 $\sim$ 7.75 hours [146], as shown in Chapter 2). Therefore, 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 [33] on current systems (see Chapter3).

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).

6.5 Increasing the Ghost Region Size

While failure masking provides scalability by itself, several application characteristics can be taken into consideration to increase its impact and, therefore, obtain higher reductions in the end-to-end execution time. Section 6.5 and Section 6.6 analyze particular characteristics of stencil iterative computations that can be exploited to that end. In both cases, the goal is to increase the time it takes for a failure in a single stencil cell to propagate and affect all other cells across the stencil system. Ideally, the impact on the simulation time of any optimization that increases failure masking probability should be minimal, but a tradeoff exists between the amount of propagation extension and its impact on the execution time. This tradeoff can be exploited depending on the application characteristics, the size of the system, and the MTBF. While presenting some related experimental conclusions in Section 4.4, we leave for future work the break-even analysis of this tradeoff depending on system size. Previous work [99] studies how to use ghost region expansion in order to reduce the latency of communications by merging different messages. This section explores the effects of communication frequency reduction achieved by an increase in the ghost region size.

6.5.1 A Guiding Example: 1-D, 3-point Stencil

Update and Failure Propagation Windows. Assuming, for example, a 1-D, 3-point stencil, if a change occurs in point p in iteration i, it will not be reflected in point $p + k$ until k iterations later (i.e. iteration $i + k$). In this case, the *update propagation window* is of k iterations.

However, since multiple points are associated with a rank (assume, P points per rank), and assuming ranks communicate every iteration, a failure in the rank r including point

p (failure happens before completing iteration i) will propagate to rank r_2 including point $p+k$ before iteration $i+k$. In particular, rank $r+1$ will stall in iteration $i+1$, rank $r+2$ will stall in iteration $i+2$, and, in general, rank $r+n$ will stall in iteration $i+n$. In the example, r_2 is $r+t$, where t is either $t = \lfloor \frac{k}{P} \rfloor$ or $t = \lceil \frac{k}{P} \rceil$. Therefore, in this example, the *failure propagation window* between point p and $p+k$ is $\lceil \frac{k}{P} \rceil$ iterations, in the best case. This is to compare to the update propagation window of k iterations.

Expanding the Failure Propagation Window. If, based on the previous example, we assume the ranks in the application communicate every two iterations instead of every iteration, the failure propagation window expands. Rank $r+1$ will be able to advance an extra iteration, and, hence, will stall in iteration $i+2$. In general, rank $r+n$ will stall in iteration $i+2\times n$. Therefore, the failure propagation window gets expanded from $\lceil \frac{k}{P} \rceil$ to $2 \times \lceil \frac{k}{P} \rceil$.

Note that the failure propagation window can never be larger than the update propagation window, as that would imply violating the semantics of the algorithm.

Delaying Communication between Ranks. In order to avoid communicating every iteration, the frequency of communication can be reduced to every several iterations. To that end, computation of each partition's bordering cells needs to be replicated in every rank that needs them.

In the simplest case of a 1-D 3-point stencil, as depicted in Figure 6.8, communication can be avoided every two iterations. Instead of exchanging a single point per border, each rank initially fetches two points per border from the corresponding neighbor and keeps them in the ghost point G_{-2} and G_{-1}. Each rank can then finish calculating the first iteration, which is done by updating cell C_0 with previous values of cells G_{-1}, C_0, and C_1, updating cell C_1 with previous values of cells C_0, C_1, and C_2, etc. Additionally, the ghost point G_{-1} needs to be updated with previous values of cells G_{-2}, G_{-1}, and C_0. After all the values are updated in the first iteration, the rank already has the updated ghost point and, therefore, is ready to perform the update for the second iteration without the need to communicate with its neighbors. Note that, since in the first iteration the ghost point position G_{-2} was not updated, the rank can not re-use it to calculate the new value of G_{-1}. Henceforth, in order to perform the third iteration, communication needs to occur, in a similar manner as

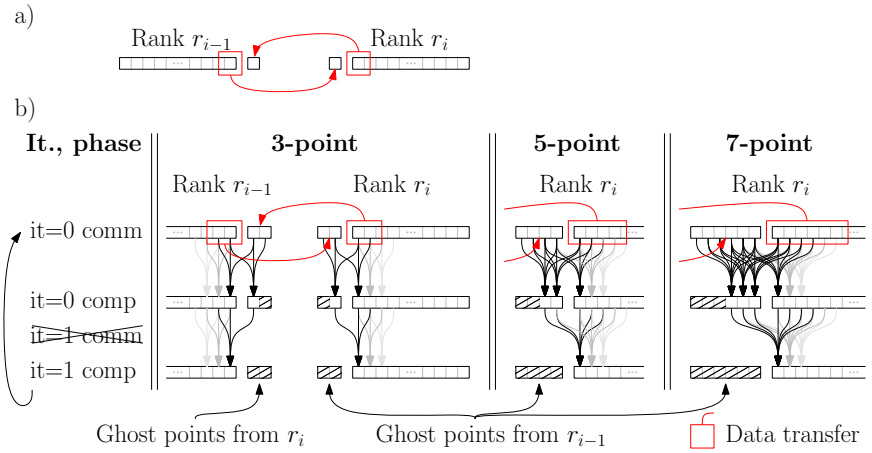


Figure 6.8: Detail of the communication and decomposition of a 1-D Stencil computation between 2 ranks. a) A 3-point calculation with standard communication pattern, i.e. one transfer at the beginning of each iteration. b) Detail of two iterations (both communication -comm.- and computation -comp.- phases) of 3-point, 5-point, and 7-point 1D Stencils. For space economy, only Rank r_i is shown in the 5-point and 7-point figures. Crossed ghost points do not contain a valid value in the specific iteration/phase.

described for the first iteration.

To summarize, in this simple 1-D 3-point stencil, by replicating the computation of one ghost point and doubling the size of the message payload, the application can double the time between two subsequent data transfers.

6.5.2 Beyond 3-point Calculations on a 1-D Stencil

More complex calculations on a particular stencil cell may require more points than the immediately adjacent neighbors. In the case of a 5-point, 1-D stencil, in order to calculate a particular cell C_i, five cells are required, C_{i-2}, C_{i-1}, C_i, C_{i+1}, and C_{i+2}. In the case of a 7-point, 1-D stencil, seven adjacent cells are required, from C_{i-3} to C_{i+3}. Therefore, to delay the communication between cores would require a higher cost due to replication than, when compared to the 3-point stencil example. However, this cost can be considered negligible compared to the cost of updating the P cells associated with each rank.

In these two more complex cases, however, the same conclusion applies: by doubling the size of the message payload, and replicating the computation of the extra ghost points (two extra points per message in the 5-point case; and three extra points in the 7-point case), the application can double the time between two message exchanges.

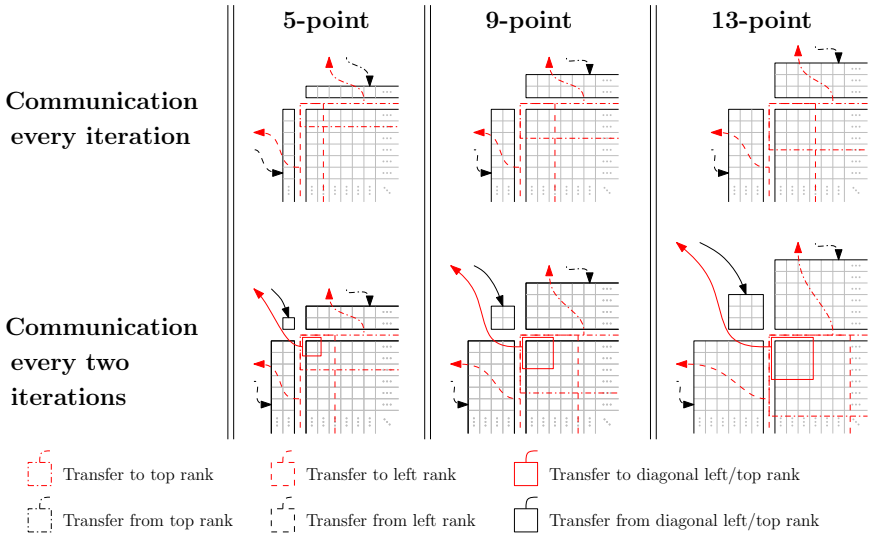


Figure 6.9: Detail of the communication and decomposition of a 2-D stencil computation between a rank and (not-shown) its left, top, and left/top diagonal neighboring ranks. The top row represents a 5-point, 9-point, and 13-point calculation with standard communication pattern, i.e. one transfer at the beginning of each iteration. The bottom row represents how the data is transferred during the communication phase –every two compute iteration, only one communication phase is performed.

6.5.3 2-D and 3-D Stencils

Extrapolating the same ideas from a single dimension to a multi-dimensional Ssencil requires understanding how the domain is partitioned and which ghost points are required for each scenario.

Two Dimensions. Figure 6.9 demonstrates that doubling the communication period requires communicating with one extra neighbor per corner in the 2-D case. Instead of communicating with four neighbors every iteration, the new algorithm must communicate with exactly eight neighbors every two iterations. In a 5-point calculation, assuming the domain size is $n \times n$, (1) communicating every iteration requires $4n$ ghost cells while (2) communicating every two iterations requires $8n+4$. In a 9-point calculation, $8n$ and $16n+16$ ghost points are required for situations (1) and (2), respectively. Finally, for a 13-point calculation, $12n$ and $24n + 36$ ghost cells are required. In general, for a $(4p + 1)$-point

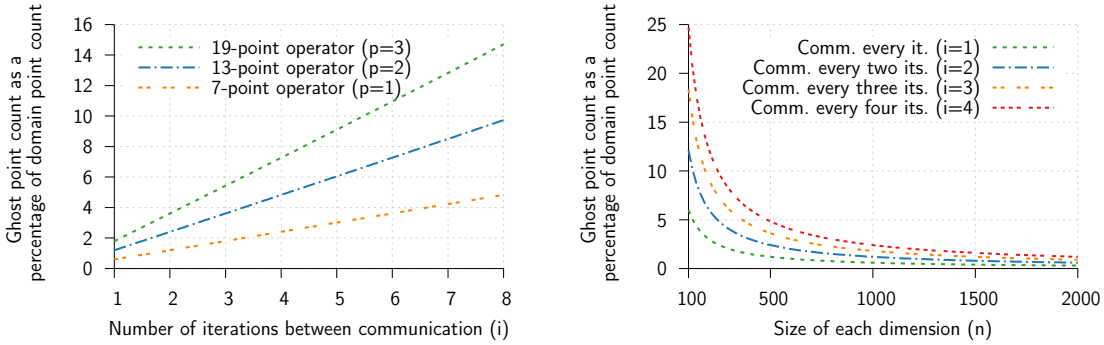
calculation,

$$
\begin{aligned}
C_1 &= 4pn \\
C_2 &= 4p^2 + 8pn \\
&\vdots \\
C_i &= 2i(i-1)p^2 + 4ipn
\end{aligned}
$$

where C_i is the number of ghost cells required for communicating every i iterations, and p is the thickness of the calculation (e.g., $p = 1$ for a 5-point calculation, or $p = 2$ for a 9-point calculation).

Three Dimensions. In the 3-D case, the domain is typically decomposed into cubes and the neighboring ranks near each face of the cube are required to update the faces. As a result, each rank communicates with six neighbors every iteration ($i = 1$). If the communication frequency needs to be reduced to two iterations ($i = 2$), each rank will require communication with $6 + 12 = 18$ neighboring ranks, as the regions in the diagonals associated with the twelve edges will be required to update the six faces. If the communication frequency needs to be further reduced (e.g., $i = 3$), the ranks with cells in the diagonals near the eight vertices will also need to be contacted by each rank because those cells will be required in order to update the cells near the cube edges. In this case, each rank will communicate with $6 + 12 + 8 = 26$ neighboring ranks. For any i such that $i \geq 3$, only these 26 neighboring ranks will be needed for ghost exchange provided that $i \leq n/p$. The latter formula assumes a three dimensional domain where each dimension is of size n. For the case where $i > n/p$, ghost regions would extend naturally into additional ranks. However, this case is not feasible in practice, since p is typically small and n is typically large in real production scenarios. Any practical reduction of communication frequency should, therefore, require interaction with a maximum of 26 ranks during the population of ghost region cells.

In a 3-D domain, the number of ghost cells required to communicate every i iterations is as follows, for a $(6p + 1)$-point calculation:



(a) Effect of the number of iterations between communication when operating on a cubical domain with side $n = 1000$.

(b) Effect of the size of each dimension of the cubical domain when using a 7-point Stencil operator.

Figure 6.10: Number of ghost cells compared to number of domain cells for different scenarios. Results were computed using the formula for C_i with different values of i and p (Figure 6.10a) and different values of i and n (Figure 6.10b).

$$
\begin{aligned}
C_1 &= 6(1)pn^2 \\
C_2 &= 6(2)pn^2 &+ \quad & (12np^2) \\
C_3 &= 6(3)pn^2 &+ \quad & 3(12np^2) &+ \quad & 1(8p^3) \\
C_4 &= 6(4)pn^2 &+ \quad & 6(12np^2) &+ \quad & 4(8p^3) \\
&\vdots \\
C_i &= 6ipn^2 &+ \quad & \frac{(i-1)i}{2}(12np^2) &+ \quad & \frac{(i-2)(i-1)i}{6}(8p^3) \\
&= 6ipn^2 &+ \quad & 6(i-1)ip^2n &+ \quad & \frac{4}{3}(i-2)(i-1)ip^3
\end{aligned}
$$

where the factor that multiplies $(12np^2)$ is the $(i-1)$-th triangular number and the factor that multiplies $(8p^3)$ is the $(i-2)$-th tetrahedral number (for $i = 1$, the factor is 0).

$n^3 + C_i$ can be used to theoretically determine the algorithmic worst-case cost of the computational portion of each iteration. It is important to note that the minimal cost of the computational portion is $n^3 + C_1$ in any case.

Figure 6.5.3 plots C_i using different values of i, n, and p. Specifically, it shows $100C_i/n^3$, the percentage of the cell points that the ghost region represents for each case, when compared to the domain region (which size is n^3). The main conclusion, which can be extracted

from Figure 6.10b, is that the overhead on the computation cost of extending the ghost region is negligible for bigger values of n. In the cases where n is small, however, the overhead of ghost region extension will be low or negligible only when the cost of communication is larger than the cost of computation. In that case, extending the ghost region implies an increase of the computation that is traded off by a larger decrease of the total communication cost.

6.6 Node-aware Mapping of Cells to Ranks

A typical approach when running an MPI application on a multi-core system is to allocate multiple MPI ranks in each socket; usually one rank per core or one rank per physical thread. As a result, a failure in any of the components shared by all cores in the socket (e.g. on-chip L3 cache, memory subsystem, network interface, power supply, or operating system) will affect multiple MPI ranks at once.

Henceforth, when trying to reduce the costs of fault tolerance, application developers and tuners need to take into consideration the underlying architecture. Specifically, this section focuses on the effect of the mapping of a decomposed stencil domain to the different available MPI ranks.

Assumptions. In this section, we assume that a particular rank is mapped into a specific socket/node statically by the MPI implementation (in this case, we will use FenixLR as presented in Section 5.3) when the MPI application is first started. Therefore, any changes require the mapping of cells to ranks by the application or a fault-aware library (in this case, the top layer within FenixLR).

We also assume a homogeneous system in which all the sockets/nodes have the same compute characteristics. Therefore, the computation time required to update a particular stencil cell or group of cells should be the same regardless of the mapping of cells to ranks. This mapping, however, may affect the network transfer costs depending on the network topology, since neighbor cells may be located in different parts of the machine depending on the mapping. The possible effects due to network characteristics will be accounted for when evaluating the approach.

Related Work. Barrett et al. [5] found that misalignment of MPI ranks to the 3D torus topology of Cray XE6 can decrease the performance of 3D stencil applications. In their study, many ghost cell exchanges were not translated to message exchange between physically adjacent nodes, as the MPI ranks were placed to shape a long rectangular geometry. This was resolved through manual rank reordering to reduce the average network hop counts of ghost cell exchange. Another rank reordering idea has been applied in the context of multicore nodes where current MPI implementations exploit shared memory copy to improve the message passing performance among the ranks placed in the same physical node. Brandfass et al. [24] demonstrated how rank reordering is able to confine interprocess communications within a single node in their unstructured CFD applications. Although this work is intended for stencil computations, we study how rank reordering can reduce the speed of propagation of the recovery delay due to message exchange.

One-dimensional Stencil Domain. A one-dimensional domain can be decomposed into different homogeneous chunks, i.e., each containing the same number of cells, and the best cell-to-rank mapping can be achieved by linearly mapping cells to contiguous ranks. Assuming a node failure, the number of cells who are affected by the failure increases by two every iteration: a failure in iteration i affects two cells when iteration $i+1$ is reached, four cells when iteration $i+2$ is reached, and so on. In this sense, it is intuitive to understand that the best mapping to minimize the recovery propagation of a node failure is the linear mapping. With a random mapping, for example, the number of cells affected by a failure increase much faster as iterations advance. Other mappings less extreme than the random mapping can be less harmful to the failure propagation time, but a linear mapping still provides the best case.

Two-dimensional Stencil Domain. In the two-dimensional case, however, a linear mapping might not be the best when trying to minimize the rate of increase of the number of cells affected by a node failure. Assuming a 16-rank node, the best mapping to minimize the propagation effect would be to assign four-by-four blocks of cells to the sixteen ranks in each node, as shown in the bottom of Figure 6.11. In the linear mapping case displayed in the center of Figure 6.11, a node failure would affect 34 cells during the first iteration, while in the quadratic mapping case, the same failure would affect 16 cells during the first

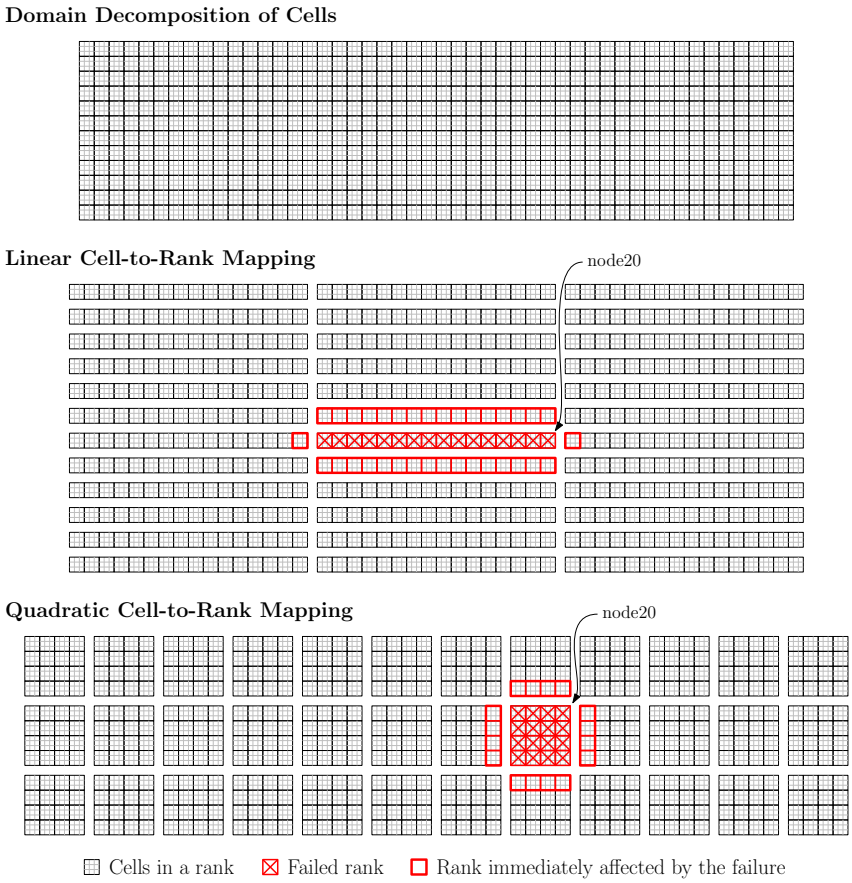


Figure 6.11: Decomposition and mapping of a two-dimensional domain into a machine with sixteen ranks per node. (top) Domain of 144×36 cells decomposed in chunks, or domain sections, of 3×3 cells each, for a total of 48×12 chunks. (center) Linear mapping of domain sections to ranks; a failure in a 16-rank node (*node20*) affects the execution of 34 neighboring ranks in the iteration immediately following the failure. (bottom) Quadratic mapping of domain sections to ranks; a failure in *node20* affects now only 16 neighboring ranks in the iteration immediately after the failure, providing a much slower failure propagation.

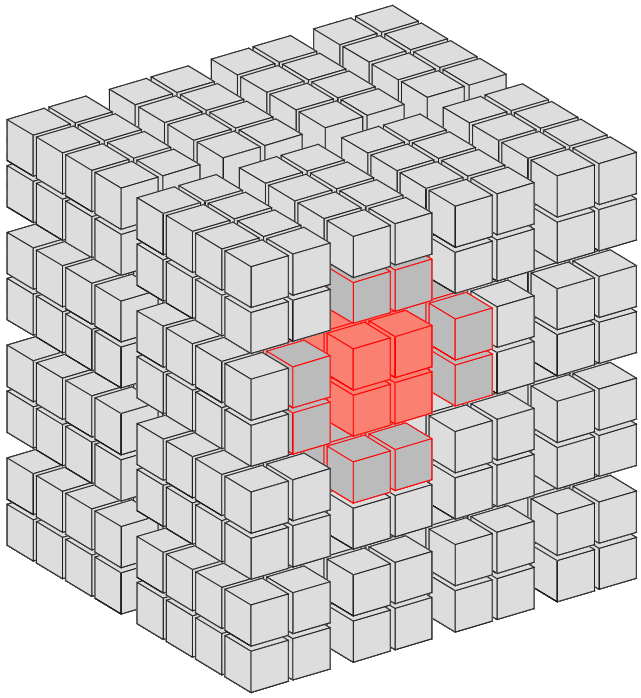


Figure 6.12: Section of a three-dimensional domain mapped to a machine with sixteen ranks per node. The mapping of domain sections to ranks is done through the shape of a rectangular prism. A failure in a 16-rank node (*node20*, set of boxes in red) affects the execution of 40 neighboring ranks (set of boxes in dark gray) in the iteration immediately following the failure.

iteration. The conclusion is that a standard, default, linear mapping –that provided an optimal failure propagation contention in the 1-D case– is not optimal in the 2-D case.

Three-dimensional Stencil domain. By default, S3D –our guiding stencil scientific computation– assigns each part of the decomposed 3-D domain following a **linear mapping**, assigned based on `MPI_CART_CREATE` using Fortran order. It assigns first cells in the X direction maintaining the Y and Z fixed; when there are no more cells in the X direction, it continues assigning cells from the line parallel to the X-axis and the next corresponding Y value; finally, when the first X-Y plane is completely mapped to the corresponding ranks, it continues filling cells from the parallel X-Y plane with the next Z value, following the same methodology. In other words, assuming that there are N_x cells in the X direction, N_Y cells in the Y direction and N_Z cells in the Z direction, and we want to map a single cell to a

rank, the rank number R would be assigned the cell (C_X, C_Y, C_Z), where

$$C_Z = \left\lfloor \frac{R}{N_X N_Y} \right\rfloor$$
$$C_X = (R - C_Z N_X N_Y) \bmod N_X$$
$$C_Y = \left\lfloor \frac{(R - C_Z N_X N_Y)}{N_X} \right\rfloor$$

This linear mapping is suboptimal, since a failure in a 16-rank node may affect 66 neighboring ranks in the first iteration after the failure.

To reduce the rapidity with which the number of affected ranks increase, we can decompose the domain into perfect cubes or use other kinds of space-filling curves. Since complex space-filling curves may imply irregular mappings (i.e. the cells assigned to a particular node do not form a homogeneous "shape" in the 3-D domain), the failure propagation speed may depend on the failed node. Since this is not a desirable property, this chapter studies how to assign cells to nodes so that the 3-D shapes of the cells are exactly the same in all nodes.

In particular, as depicted in Figure 6.12, for a machine with sixteen ranks per node, a $4 \times 2 \times 2$ **rectangular prism** (in any direction) homogeneous mapping minimizes the rapidity of failure delay propagation to 40 neighboring ranks in the first iteration after the failure.

We can generalize the rectangular prism mapping depicted in Figure 6.12 as follows. Making the same assumptions as before, as well as the fact that the rectangular prism shape must be $G_X \times G_Y \times G_Z$ (for example, $4 \times 2 \times 2$ in the figure), with a size of $S_G = G_X G_Y G_Z$ ranks, the rank number R would be assigned in the core number (Co_X, Co_Y, Co_Z) of the compute node with coordinates (Cn_X, Cn_Y, Cn_Z), and would be mapped to the cell

(C_X, C_Y, C_Z), where

$$
\begin{aligned}
Cn_Z &= \left\lfloor \frac{R}{G_Z N_X N_Y} \right\rfloor \\
Cn_X &= \left(\frac{R}{S_G} - \frac{Cn_Z N_X N_Y}{G_X G_Y} \right) \bmod \frac{N_X}{G_X} \\
Cn_Y &= \left(\frac{R}{S_G} - \frac{Cn_Z N_X N_Y}{G_X G_Y} \right) \Big/ \frac{N_X}{G_X} \\
Co_Z &= (R \bmod S_G) \,/\, (G_X G_Y) \\
Co_X &= ((R \bmod S_G) - (Co_Z G_X G_Y)) \bmod G_X \\
Co_Y &= ((R \bmod S_G) - (Co_Z G_X G_Y))/G_X \\
C_X &= (Cn_X G_X) + Co_X \\
C_Y &= (Cn_Y G_Y) + Co_Y \\
C_Z &= (Cn_Z G_Z) + Co_Z
\end{aligned}
$$

6.7 Experimental Evaluation

This section presents the experimental evaluation performed to determine the performance and effectiveness of the local recovery and failure masking approaches (as described in Section 6.2, modeled in Section 6.3, and evaluated in Section 6.4), as well as ghost region extension and rank remapping that target to enhance the probability of failure masking (as described in Section 6.5 and Section 6.6, respectively). The experimental evaluation is based on the S3D combustion simulation, a large scale stencil computation, running on top of the Titan Cray XK7 supercomputer at ORNL.

This evaluation demonstrates that the algorithms implemented in FenixLR, as presented in Section 5.3, efficiently tolerate multiple process, node, and multinode failures occurring at a wide range of frequencies. It also demonstrates that the total recovery overhead can be reduced due to multiple failure masking. In general, this section shows that, even though the overhead of globally recovering from N independent failures is in the order of $N \times O_1$ (where O_1 represents the average overhead of recovering from a single failure), the total overhead is closer to O_1 when the same N failures are recovered locally. This section shows

the benefits of FenixLR's local recovery capabilities using S3D on up to 140736 cores (140608 + 128 spare cores). It is experimentally demonstrated how in real scenarios, local recovery can mask failures, i.e., result in a total overhead that is comparable to one failure recovery, regardless the number of failures.

Chapter 4 has used a Fenix implementation on top of a ULFM prototype to successfully recover from failures occurring as frequently as every 47 seconds. Chapter 5 has demonstrated how the FenixLR implementation reduces sources of overhead and offers optimized recovery constructs able to tolerate failures occurring as frequently as every 5 seconds.

6.7.1 Experimental Evaluation Goals

As this section discusses the experimental evaluation of failure masking, the following are the main goals to be addressed: (1) show that, by using the presented failure masking methodologies, stencil computations such as the tightly-coupled S3D combustion simulation can tolerate failures coming at high rates, (2) establish 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), and (3) demonstrate, experimentally, that the probability of multiple failures masking each other can be increased by application-aware algorithmic changes such as ghost region expansion and cell-to-rank mapping.

Following subsections present the methodology of experimentation as well as describe the experiments in detail.

6.7.2 Experimental Methodology

A key goal of this evaluation is to study how the presented approach behaves at current scales, and use this to explore behaviors and performance due to future extreme-scale failure rates. As a result, we have conducted our experiments on up to 140608 cores. As mentioned above, all the experiments were performed on the Cray XK7 Titan at ORNL.

The evaluation first experimentally demonstrates the full benefit of local recovery capabilities in FenixLR: its ability to mask multiple failures. To that end, using S3D on up to 140608 (plus 128 spare ranks), the experiments demonstrate how real node failures can mask each other, obtaining a similar overhead regardless of the number of randomly injected

failures.

The section ends by studying how the techniques studied in Section 6.5 and Section 6.6 can increase the probability of failure masking.

All the aforementioned experiments inject node failures, which are simulated by determining all application processes in execution in a particular node and sending simultaneous `SIGKILL` signals to all of these processes. As the network setup parameters are stored in process memory, when killing the processes no software disconnections are allowed – this is consistent with the behavior if a real node failure occurs. Processes on other nodes will receive error codes when trying to perform a uGNI operation with the processes that failed. In what follows, 'failures' refer to '*node* failures', which is equivalent to *N-process* failures, where N is the total number of processes on a system node, blade, etc. By default, the experiments use $N = 16$. All the experiments present the average, first and third quartile, maximum, and minimum of five repetitions, unless otherwise specified.

As mentioned above, it has been observed that, statistically, failures tend to cluster together in time, appearing as periods of time with high instability separated by more stable periods [146]. For example, even though Titan's MTBF is approximately 7.75 hours, around 30% of all failures occur within one hour of a previous failure. The goal of these experiments is to focus on extreme cases in which unrelated failures occur within short periods of time. In particular, we are interested in those periods of high instability that may occur on future extreme scale machines. As failure distribution predictions and MTBFs of future extreme scale systems vary significantly, this section presents a worst case study and, therefore, focuses on experiments with failures frequencies below one minute. If we consider lower failure frequencies in the order of tens of minutes (or even hours), negligible overheads can be observed due to failures when recovered using the presented techniques. Specifically, long simulations experiencing low failure frequencies observe: (1) negligible aggregated checkpointing cost due to its low overhead and relatively low frequency, (2) rollback overhead similar to that of checkpointing (assuming checkpoint frequency is adjusted correctly following Daly's approach [41]), and (3) negligible process recovery cost (as shown by Figure 5.7).

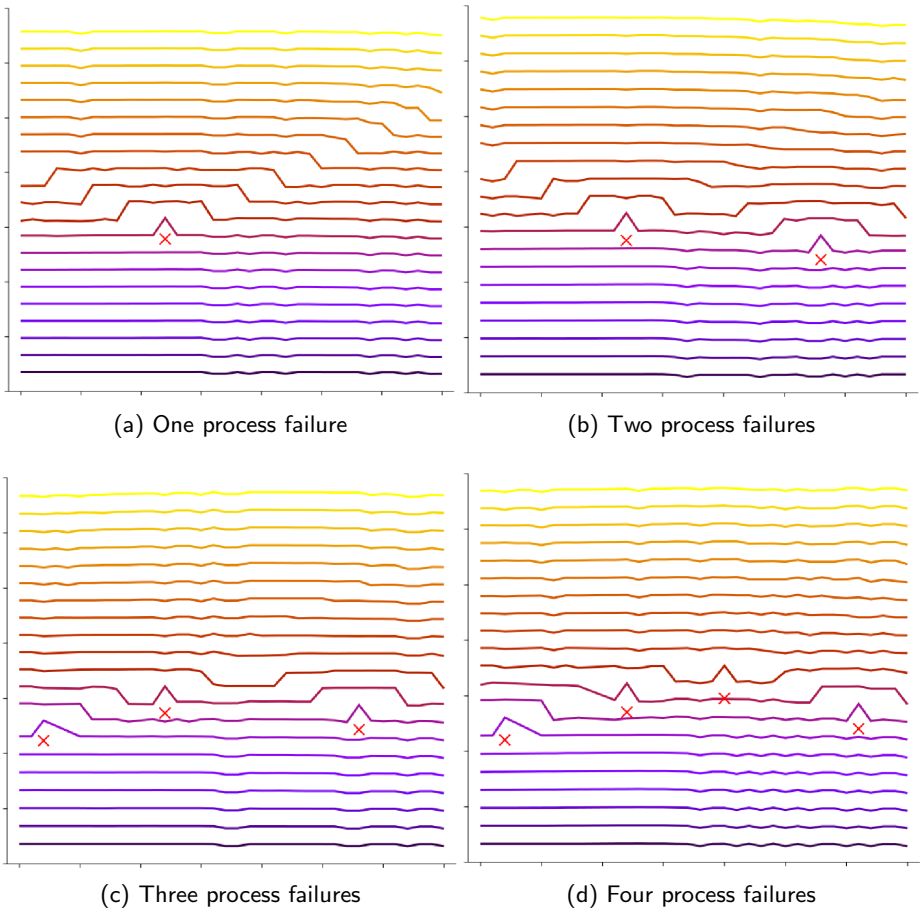


Figure 6.13: Behavior of local recovery for a 1D PDE using 36 cores (32 compute cores and 4 spare cores). X axis represents process number (or rank) and Y axis indicates wallclock time. Each line in a figure represents a timestep, and the color of the line represents how advanced the simulation is (i.e., it advances from dark purple to yellow). 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. Instead, 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. Gamell et al. [68].

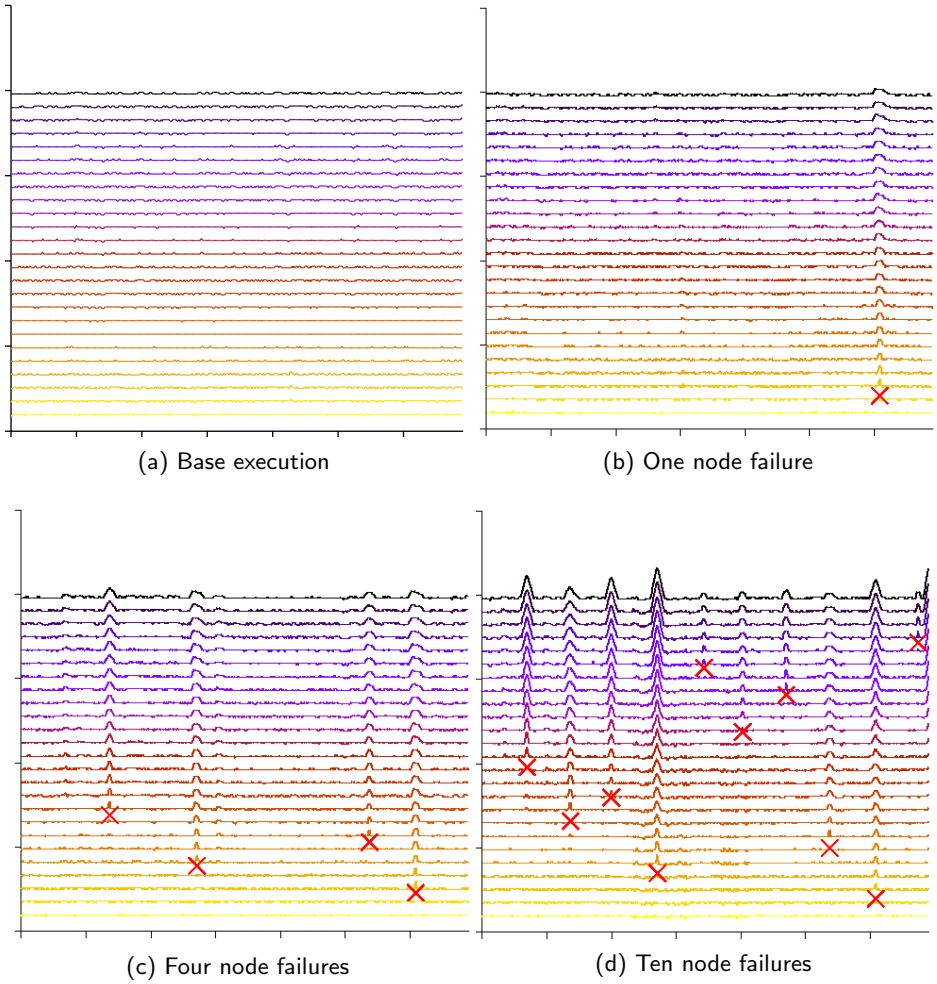


Figure 6.14: Behavior of local recovery for a 1D PDE using 13984 cores (13824 compute cores and 160 spare cores), with failures injected every 10 seconds. ©2015 ACM (reprinted with permission) Gamell et al. [68].

6.7.3 Experiments using a 1D PDE

The first experiment tries to study the behavior of local recovery for a one-dimensional stencil-based Partial Differential Equation (PDE) solver, or 1D PDE for short. Figure 6.13 shows several executions of 1D PDE on 32 compute cores and 4 spare cores with different number of failures. These figures are based on results obtained from actual runs on Titan (Cray XK7 at ORNL) during which we injected real process failures as described above. The X axis in the plots represents the core number (or rank number), and the Y axis represents wall clock time. Each rank simulates a certain number of points in the 1-D domain. Adjacent ranks operate on adjacent spans of the 1-D domain. Each horizontal line in a plot represents an iteration (i.e., every time the solver communicates with the neighbors in order to advance the solution). Red crosses represents injected process failures. A straight line means that all processes complete the iteration at the same physical time. When a failure occurs, the recovery delay does not get propagated immediately to the entire domain, but, instead, the neighbor processes that are immediately adjacent to the recovered process are the first to be delayed. This delay propagates out to their neighbors in the next iteration, and their neighbor's neighbors in the following iteration, and the process repeats until the entire domain is eventually covered. Note how the two, three and four failure cases shown in Figures 6.13b, 6.13c and 6.13d have a similar recovery overhead as the one failure case, i.e., Figure 6.13a.

Figure 6.14 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 delay propagation waves caused by the failures do not merge. As a result, in this case the total time to solution only increases by the time to recover from a single failure, independently of the total number of failures.

By comparing Figure 6.13 with Figure 6.2 (left) and Figure 6.14 with Figure 6.2 (right), it can be seen that results from the model and simulation accurately predict the real experimental results. This implies that the presented discrete event simulator does capture the benefit of local recovery. The results presented above not only validate our algorithm and implementation, but also demonstrate that local recovery can be beneficial in extreme scale environments, where high-frequency failures are expected. The results also demonstrate

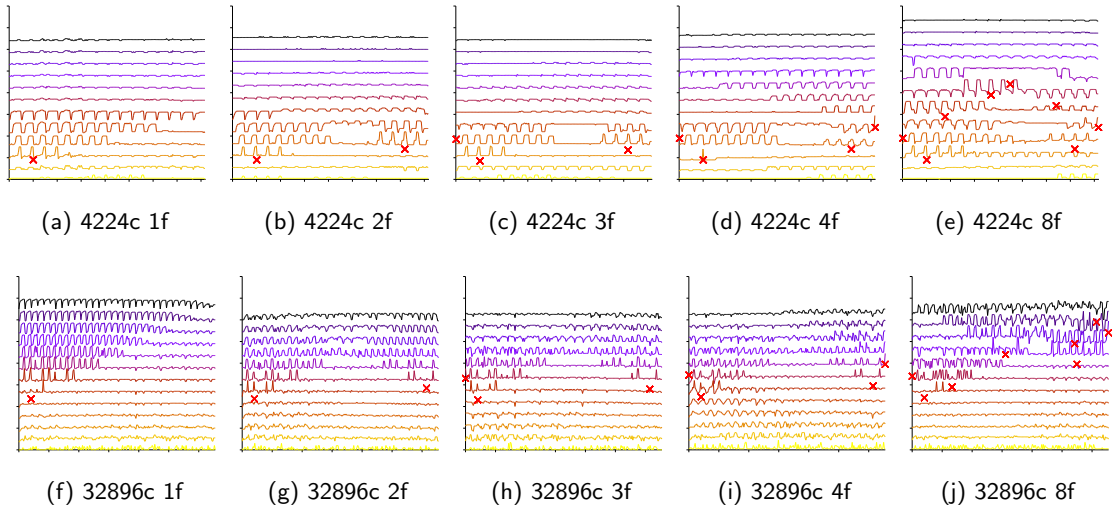


Figure 6.15: Execution profile of S3D while injecting different number of failures empirically demonstrating the existence of the failure masking effect. Figures on the top row represent tests with one, two, three, four, and eight node failures running on 4224 cores, corresponding to 4096 compute cores (an S3D domain decomposed in a grid of 16^3 cores) and 128 spare cores. Figures on the bottom row represent the same tests running on a larger domain with 32896 cores, corresponding to a 3-D grid of 32^3 as well as 128 additional spare cores. In each figure the x-axis represents the core number (as MPI ranks in the world communicator) while the y-axis represents the walltime, advancing from the start of the application to its end. Each line represents the time a particular core finishes computing a particular iteration. Note that failures, denoted by red crosses, are recovered and rolled back locally, which translates to a delay that is propagated throughout the domain in successive iterations. The end-to-end time when injecting eight failures is slightly longer than in the other cases. In all other cases, the end-to-end time is similar, demonstrating the benefits of failure masking. Note that ghost resizing or rank remapping have not been applied in these experiments, motivating the need for these techniques to achieve slower propagations.

that local recovery is a scalable approach, both in the number of failures and the size of the system.

6.7.4 Experiments using S3D

Figure 6.15 shows the behavior of local recovery for the S3D Stencil 3-D PDE solver. Each line in this figure represents a timestep, and the color of the line represents how advanced the simulation is – it advances from yellow to dark purple. The X axis in the plots represents the rank number, which is linearly mapped to the 3D domain in S3D. This mapping between 3D space and 1D ranks is done in a straightforward manner – beginning at 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. The Y axis represents wall clock time. In the caption for each plot, a 'c' refers to cores and an 'f' refers to the number of node failures injected. Note how the overheads in the first three columns are the same, which empirically demonstrates how failure masking works. The plots in these figures follow the same format as Figure 6.13. In the 3D case, however, the communication pattern between ranks is not as obvious in the plots as in the 1D case due to the mapping of ranks to the 3D domain. Stencil-based communications in the 3D domain proceed as follows: each process (except along the edges) communicates with its six neighboring processes in the up/down, front/back and left/right directions.

All experiments have allocated 128 spare ranks and either 4096 or 32768 ranks. Failures counts ranging from 1 through 8 have been injected to study how the propagation of the recovery is spread across the nodes as well as how the recovery delay propagation wave due to different failures interact, in the case of S3D. It is important to note that, in all cases, the total time to solution (the height of the uppermost iteration) is similar regardless of the total number of failures. This holds in all cases except the execution of 4096 ranks while injecting 8 failures, since one of the failures strikes in a node that have been already delayed by a previous failure.

Figure 6.16 summarizes the relative costs of recovering from different numbers of failures (from 1 to 8) at different scales (4096, 8000, 13824, 32768, 64000 and 140608 cores). Each bar represents the average of five executions, except for experiments larger than 64000 ranks due to allocation issues. These issues also affected scheduled experiments with larger failure counts. The Y axis represents the execution time relative to the average overhead of suffering a single failure. Therefore, the figure depicts how the overheads of different failures are masked, showing how comparable is the total overhead of a particular test to that of a single failure. In most cases, the overheads are very similar, varying around 2% or lower in cases where failures are completely masked. The variability is due to the fact that the rollback overhead is the main factor in the recovery process and it depends on the distance of a particular failure to the last checkpoint. A failure occurring right after a checkpoint will have minimal rollback overhead while a failure occurring right before a checkpoint will

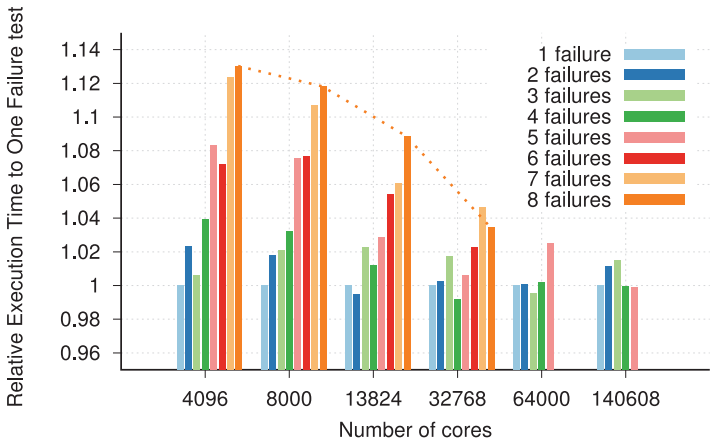


Figure 6.16: End-to-end execution time of S3D while injecting multiple node failures and recovering locally, relative to the end-to-end execution time when injecting a single failure. A relative time similar to a unit represents failures masked perfectly (variability is due to variable rollback overheads), while relative times in the order of 1.06 or even 1.10 indicate that not all failures masked each other, but some failures occurred after the delay already propagated to that node. Note how increasing number of nodes implies a decrease of total overhead with high failure counts (e.g. eight failures, as shown with the trend line), indicating that an increase in core count increases failure masking probability. ©2015 ACM (reprinted with permission) Gamell et al. [68].

have a rollback overhead practically equivalent to recomputing an entire iteration one more time. In cases where this relative overhead increases up to 6%, or even 12%, at least one failure occurred in a node after it had been delayed by the recovery of a previous failure. An example of this can be seen in Figure 6.15, where, as mentioned before, comparing Figure 6.15e with Figure 6.15a, Figure 6.15b, or Figure 6.15d shows a non-negligible increase in the total time to solution due to a failure occurring after the propagation of a prior failure. In this specific case, the node experiencing the third failure (counting from the left) has experienced a delay due to the second failure.

Results in Figure 6.16 and Figure 6.15 show, therefore, that benefits of failure masking can be of critical importance when trying to minimize the overheads of recovery.

The results presented in Section 6.7.3 and Section 6.7.4 have demonstrated how local recovery enables failure masking, which is also highly desirable at extreme-scales. Specifically, it has been 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).

6.7.5 Increasing the Failure Propagation Window on S3D

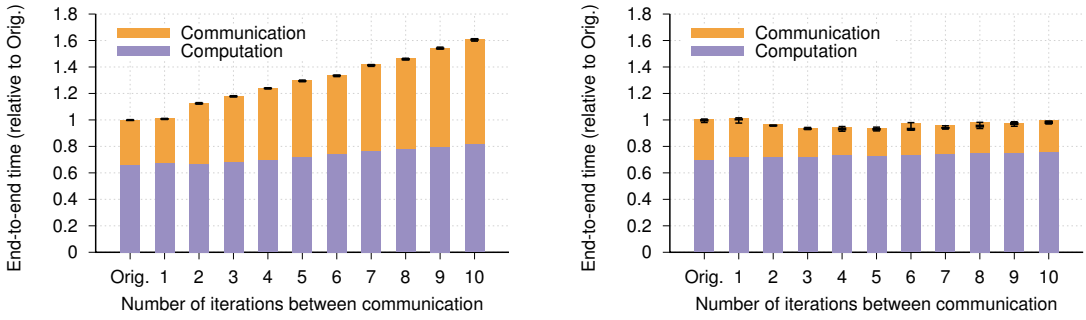
The goal of this experiment is to demonstrate that by exploiting the characteristics described in Section 6.5 and Section 6.6 the opportunity for masking failures is increased. To this end, the current set of experiments shows how the failure propagation window can be expanded by increasing the ghost region size and re-mapping cells to ranks. As described before, the failure propagation window is the pace at which the number of ranks affected by a particular failure increases.

The failure propagation window closes when the effect of recovering from a failure reaches all the ranks in the machine and can be measured (1) generically using number of iterations computed since the iteration in which the failure occurred or (2) specifically using any measure of time. Unless otherwise indicated, the results in this part of the evaluation section use the former, more generic, method for measuring the failure propagation window.

We augmented the S3D main code loop to extend the communication frequency by increasing the ghost region size and replicating part of the ghost region in two different ranks. We also augmented the part that decomposes and maps the simulated domain in the different ranks to support the topology described in Figure 6.12.

Evaluating the Overhead of Ghost Region Expansion. To evaluate the overhead of the ghost region expansion technique on the execution time we performed a set of experiments using the unmodified S3D as well as the augmented code. Figure 6.17 shows the results of different levels of ghost region expansion, starting from a single layer of ghost points and increasing it to allow communication up to every ten iterations. As the computation and the communication are overlapped by using asynchronous operations, the portion of each bar labeled 'Communication' represents the time spent in relevant MPI operations.

Figure 6.17a was created performing 100 iterations of S3D, parameterized as in the rest of the experiments in this section. The results show an increasing overhead due to ghost region expansion. In particular, since communicating every iteration requires message exchange with 6 neighbors and communicating every two iterations requires contacting 18 neighbors, the communication time is significantly larger in the latter case. The figure shows how the impact of decreasing the communication frequency to once every three or four iterations is below 25%.



(a) S3D execution

(b) S3D execution modified to operate on one floating-point number per cell, instead of nine.

Figure 6.17: Overhead of different levels of ghost region expansion on the end-to-end time when compared to the unmodified, original S3D code (labeled as 'Orig.' in both subfigures). The X-axis represents the number of iterations between consecutive communications (parameter i in the model presented in Section 6.5). The experiments were conducted using a generic Linux cluster of 32 nodes connected via Infiniband. Each node has two Intel quad-core Xeon E5620 processors and 24GB of RAM. Our executions used a 27-node allocation (running 8 processes per node) to simulate a cubical domain split into $6^3 = 216$ homogeneous partitions. Each bar represents the average of 10 repetitions, each simulating 100 S3D iterations. No checkpoints are created nor failures injected in any of these tests.

A similar set of executions that use and communicate a single real number per cell (rather than nine real numbers per cell) is shown in Figure 6.17b to exemplify a slightly different application pattern. In this case, it can be observed that ghost region expansion could actually benefit execution time rather than negatively impacting it. In particular, trading off extra computation per iteration (by increasing ghost region size) in order to decrease communication frequency offers an overall gain in performance.

As a conclusion, the executions depicted in Figure 6.17 demonstrate the importance of understanding and studying the aforementioned tradeoff in each individual stencil application. Particular application behaviors influence the optimal communication frequency and the impact it has on both computational performance and fault tolerance overhead.

Comparing Different Domain Cells to Rank Mappings. The first test needs to determine the effect of the domain cell to rank mapping on the propagation delay shape. To that end, Figure 6.18 shows different mapping strategies on a 512-rank execution with a failure injected in sixteen ranks. The worst behavior can be observed when randomly assigning the cells to the ranks. The figure shows the behavior of two random mappings. Figure 6.18 also depicts tests done with two regular configurations: cubes of side two and

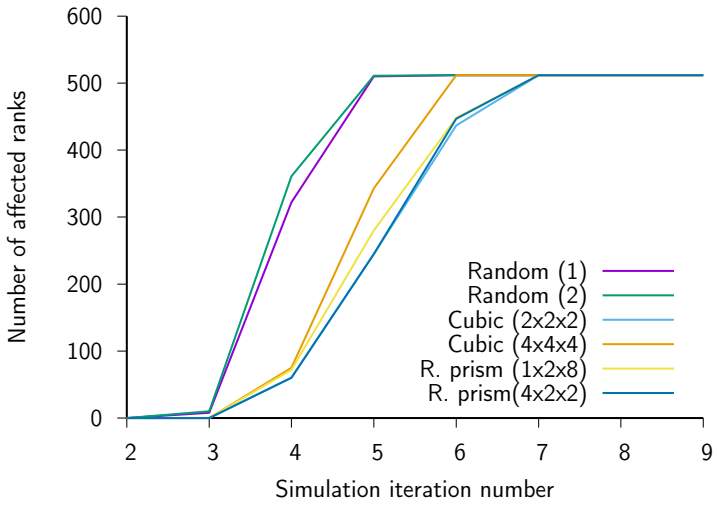


Figure 6.18: Different domain cell to rank mapping strategies on a 512-rank execution with a failure injected in all sixteen ranks of the third node, on the 16th second after the starting of the application. Two random mappings are tested, providing the worst behavior, as well as two cubic configurations ($2 \times 2 \times 2$ and $4 \times 4 \times 4$) and two rectangular prism configurations ($1 \times 2 \times 8$ and $4 \times 2 \times 2$). Note that only the rectangular prism configurations fill exactly a 16-rank node, which is the target architecture.

cubes of side four, both providing power of two total number of cores – and, hence, divisible by the total domain size of 512. Either configurations are suboptimal, since the total number of ranks included in such cubes are not the exact number of rank in a node of Titan. Therefore, two extra configurations based on rectangular prisms are tested. The sizes of these two extra configurations are $1 \times 2 \times 8$ and $4 \times 2 \times 2$, and in both cases the total number of ranks per decomposed shape is exactly sixteen, which matches the target architecture. The results show that the rectangular prism of $4 \times 2 \times 2$ and the cubic shape of side 2 have a similar propagation delay curve. The remaining experiments with rank re-mapping use the rectangular prism of $4 \times 2 \times 2$, unless otherwise indicated.

Ghost Region Extension and Rank Re-Mapping Empirical Evaluation. Figure 6.19 shows several executions of S3D on 4096 cores, in which a single node failure was injected by killing the 16 ranks running on it. In particular, all the processes in node 31 received a `SIGKILL` at the 25th second from the beginning of the execution. On the X-axis, the iteration number is plotted while the Y-axis represents the total number of MPI ranks affected by the recovery overhead of that failure. The number of ranks affected are counted as depicted in Figure 6.20. The left subfigure of Figure 6.19 shows the profile of

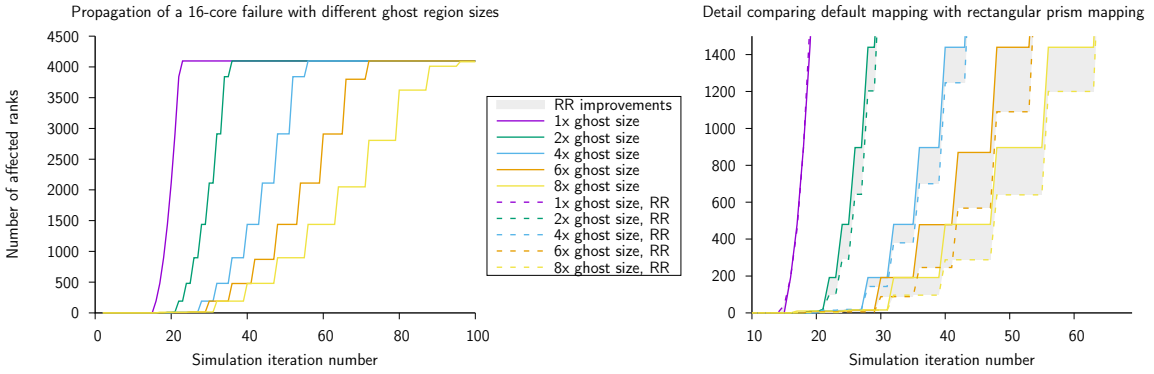


Figure 6.19: Effects of ghost region expansion and rank remapping on the propagation window. Each line shows an execution on 4096 ranks in which a node failure (16 cores) is injected. The left figure includes experiments with increasing number of ghost region sizes with a default cell-to-rank mapping. The right figure includes a detail of the same executions with both the default mapping and a mapping using the rectangular prism approach. The gray area represents the improvement in the propagation curve induced by using the topology-aware mapping. Each line is one of four executions; the difference between executions was unnoticeable.

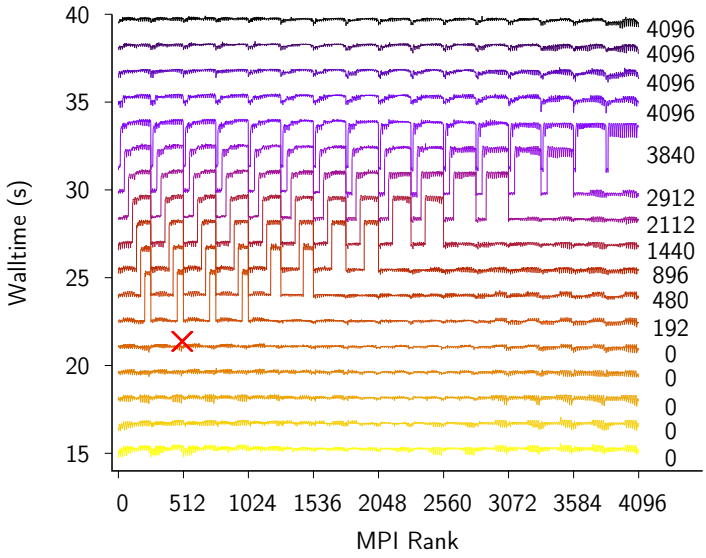


Figure 6.20: Detail of the execution from Figure 6.19 using the default mapping and no ghost region extension, showing how the ranks affected by a failure are counted. The X-axis depicts MPI rank number. The Y-axis depicts the walltime, and each line represents the point in time in which an S3D iteration is finished by each rank. The number on the right of each line counts how many ranks have been affected by the node failure indicated with a red cross.

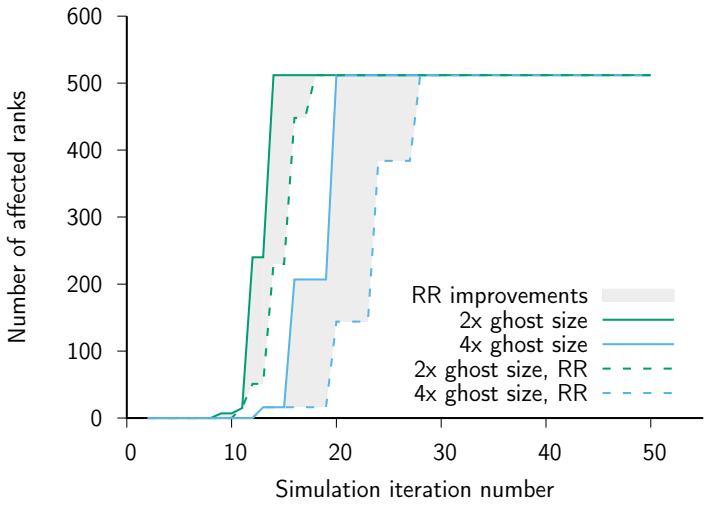
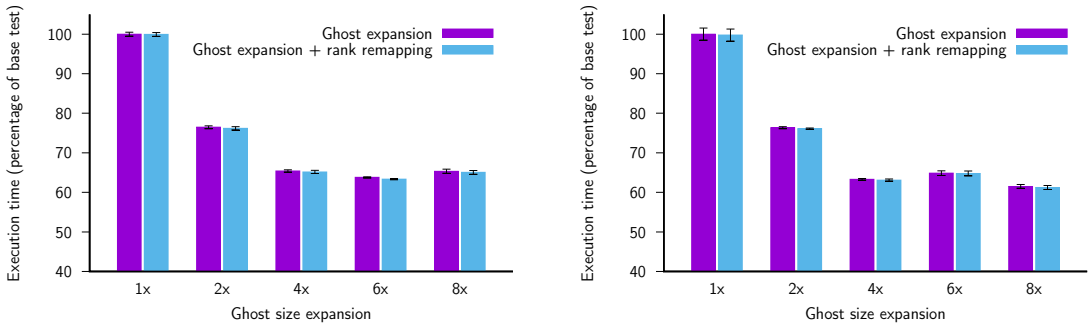


Figure 6.21: Effect of ghost point size and rank re-mapping on the total number of affected ranks by a node failure on a 512-rank execution. The node failure is injected in all cases by injecting a 16-rank failure in the third node at the 16th second. The gray area represents the improvement in the propagation curve induced by using the topology-aware mapping. Each line is one of four executions; the difference between executions was unnoticeable.

propagation of the affected ranks for different ghost sizes, from the base case of one to the extreme case of eight. In each case, a test labeled 'Gx ghost size' represents a test in which the communication is done every G iterations. The scale of these experiments being small (4096 cores), at least when compared with production runs, for the base case of no ghost region extension, the failure is propagated to all the ranks in the domain quickly: in less than 5 iterations. When extending the ghost region to as low as twice the original size, this number is extended to 16 iterations, and in the extreme case of eight times the ghost size, the propagation delay is extended to 80 iterations. On the other hand, the right part of the same figure shows a zoomed detail of the same five cases. Each of the cases was repeated with a topology-aware $4 \times 2 \times 2$ rectangular prism mapping (as shown in Figure 6.12). The re-mapping tests are marked as 'RR' and plotted in a dashed line. The respective improvements of re-mapping the ranks in each of the ghost size extension cases are depicted as a gray area between both dashed and non-dashed lines. In all cases, especially with higher ghost size ranges, re-mapping the ranks as a rectangular prism as opposed to the default linear mapping offers a delayed propagation of the failure, which is an extremely desirable effect. In all ten cases, each line is one out of four executions: the reason why only one is included is due to unnoticeable difference between the different executions –i.e.



(a) 4096 ranks (b) 32768 ranks

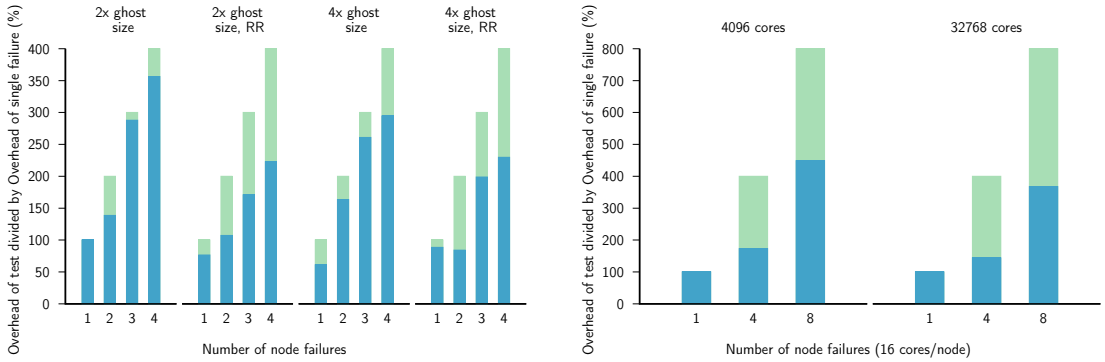
Figure 6.22: Execution time of the different techniques while injecting and recovering from a single node failure. The base test is considered the left-most bar in each of the figures, representing no ghost expansion (1×) and no cell to rank re-mapping. The y-axis represents the total time, as a percentage compared to the base test. The number of iterations between consecutive checkpoints have been set to follow the degree of ghost rank expansion. Aside from the ranks indicated in the captions, the experiments allocated an additional set of 128 spare ranks.

the lines almost overlap.

A similar experiment has been repeated with a much smaller domain of 512 total ranks, and is depicted in Figure 6.21. In this case, a node failure has been injected in the 16th second (all sixteen ranks running in node 3 receive a `SIGKILL` at the 16th second).

Additionally, the performance of the techniques described in Section 6.5 and Section 6.6 are studied by this empirical evaluation while injecting a single node failure. Figures 6.22a and 6.22b depict the total, end-to-end, execution time when using 1×, 2×, 4×, 6×, and 8× ghost region expansion combined with both rank re-mapping and no re-mapping, while running on 4,000 and 32,000 ranks. These experiments are performed using a cube S3D domain decomposed on $16 \times 16 \times 16$ ranks and $32 \times 32 \times 32$ ranks, respectively. The checkpoint period has been adjusted to coincide with the ghost region expansion rate, since checkpointing involves communication with a pre-determined neighboring rank, and this communication needs to be delayed to maximize the failure delay propagation window. Therefore, the increase in performance when expanding the ghost region and further delaying communication is both due to the savings from reducing communication latency and synchronization as well as reduced usage of bandwidth from the checkpointing process.

As can be observed in both cases, the use of rank re-mapping has no performance penalty, while greatly improving the probability of failure masking, as expected and shown



(a) Two and four times ghost region expansion; default and rectangular prism rank mapping techniques. All executions run S3D on 512 ranks with an additional 64 spare ranks. Overheads are compared with the single failure, "2x ghost size" test.

(b) No ghost region expansion; $4 \times 2 \times 2$ rectangular prism rank mapping. The subfigure on the left used S3D with 4096 compute ranks and 128 spare ranks, and the subfigure on the right used S3D with 32768 compute ranks and 128 spare ranks.

Figure 6.23: Overhead on the total time to solution over several executions with increasing number of failures recovered using local recovery in different scenarios. The X-axis for each subfigure is the total number of failures, while the Y-axis represents the overhead of each test relative to the overhead caused by a single failure (%). For each total number of failures, the extrapolated overhead in a theoretical execution with best-case global recovery (100% for one failure, 200% for two failures, 300% for three...) is indicated by the lighter color (green). As such, the lighter part of each bar indicates overhead reductions due to failure masking.

previously in the right part of Figure 6.19.

Staircase Behavior of Ghost Region Extension. Note that, in both Figure 6.19 and Figure 6.21, the cases with extended ghost region present a staircase shaped profile, which is more pronounced the larger the extension. This shape can be explained by understanding that during the horizontal part of each 'stair', the delay of recovery is not propagated for several iterations. The reasoning behind the periods of non-propagation lies in the fact that, by increasing the ghost region size G times, the communication among ranks will only occur every G iterations, and the delay is only propagated while communication occurs.

Total Overhead for Different Number of Failures. Figure 6.23a depicts the total overhead of several failures when compared to the overhead of recovering from a single failure. Each subfigure represents a different combination of ghost region size increase and rank re-mapping technique, namely 2x and 4x ghost sizes each with both a default linear rank mapping and a $4 \times 2 \times 2$ rectangular prism mapping. For each failure count, Figure 6.23a

compares the total overhead with local recovery of real executions with a theoretical, best-case global recovery. The comparison with global recovery in these Figures is considered to be best-case since the recovery cost (excluding the rollback cost) is considered to be identical in both local and global recovery, which is not a safe assumption even at medium scales of hundreds or thousands of cores. We can observe that when jumping from 2 to 4 ghost region size extension multipliers the opportunities of failure masking are increased and, therefore, the total overheads are lower in the case of three and four node failures. The same happens when changing the mapping strategy from linear to based on a $4 \times 2 \times 2$ rectangular prism, in both the cases of 2x and 4x ghost regions sizes. The results show only a single experiment for each test.

In Figure 6.23b the same overhead and comparison with global recovery that was depicted by Figure 6.23a can be observed. In this case, the scale is increased from 512 ranks to 4096 and 32768, respectively. Local recovery offers a much lower overhead than global recovery, as seen in the 512-rank case, which can be explained through the benefits of failure masking. It can be seen how the local recovery version offers a higher reduction in the 32768-rank execution than in the 4096-rank case, which is expected, since the possibilities of failure masking are increased in larger domains.

Chapter 7

Conclusion

7.1 Conclusion

Science and engineering applications increasingly rely on very large-scale simulations to make advances, facilitate discovery, and inspire deeper understanding in their respective fields. The expectation of such advancements motivates the need to achieve exascale computing capability by early next decade. The enormous scale and complexity of an exascale system, which will most likely 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, process and node failures are observed, on average, every few hours, and it is expected that the reliability of future HPC systems will decrease.

With exascale systems, the expected 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, establishing fault tolerance techniques as a necessity. Hardware-level fault tolerance has clear advantages, such as allowing the software to assume a reliable substrate. To counter, the abstraction of a failure free machine may not be sustainable due to its elevated implementation and maintenance costs. In order to efficiently tolerate frequent failures, different lines of research have suggested the introduction of significant algorithmic or even programming model switches away from traditional SPMD (single program, multiple data) approaches. Since most existing simulations use (and are proven to be accurate with) an SPMD model and are composed of hundreds of thousands of lines, making such drastic changes to the model is costly. As an alternative, HPC centers recommend that users periodically create either application-agnostic or application-specific checkpoints and store them in the parallel filesystem (PFS), assumed to be resilient due to techniques such

as data replication. Once a failure occurs, surviving application processes must be stopped and the application can be restarted from the last checkpoint. This technique is known as PFS-based checkpoint and restart (C/R).

This dissertation presents the technique of on-line recovery through application-awareness. In particular, this dissertation presents and discusses the advantages and limitations of global and local recovery, as well as describes how to enable multiple failure masking in highly unreliable resources. As a part of this discussion, the Fenix and FenixLR frameworks are presented as an attractive and viable alternative to PFS-based C/R, having better performance and resilience tradeoffs. Through Fenix and FenixLR, this dissertation describes, implements, models, and evaluates online and application-aware resilience approaches for SPMD and MPI applications. Application-aware online recovery addresses some of the shortcomings of traditional PFS-based C/R. It eliminates the overhead associated with process restart as well as allows an application's memory space to survive failures and be used for either the more efficient reconstruction of application state or the optimization of checkpoint storage.

In this dissertation we have explored an approach for online, transparent recovery from high-frequency process, node, blade, and cabinet failures in MPI-based parallel applications using application-guided checkpointing as a data resilience method. Also presented are the design and implementation of this approach within the Fenix framework, supplemented by the deployment of Fenix on the Titan Cray XK7 production system at ORNL. With the addition/alteration/rearrangement of less than 35 lines of code, Fenix was able to introduce resilience into the S3D combustion application. We also presented an extensive experimental evaluation of Fenix on Titan using S3D while injecting real failures. These experiments demonstrate that the combination of techniques implemented in Fenix provide a viable solution for addressing failures at extreme scales. The viability of Fenix as a solution is only further evident when considering that the observed 18% overhead is lower than that which can be observed in current S3D large-scale production runs experiencing 9 failures per day on Titan. The scalability of each stage of failure recovery in Fenix was evaluated, concluding that data checkpointing, data recovery, and process recovery scale well with an

increase in the number of cores.

This dissertation has also presented the advantages, most prominently in the area of scalability, of application-aware checkpointing to support transparent node failure recovery and demonstrates its ability to eliminate the cost of coordination under certain assumptions. Implicit coordination allows diskless, neighbor-based checkpointing to scale up to 250,000 cores and shows a 300%-400% benefit compared to blocking coordinated checkpointing, even in a highly balanced application like S3D. This reaffirms what recent studies suggest [79]: the vision of a failure-free machine will not be sustainable in future systems and application-aware resilience techniques will likely be required at exascale.

Certain applications, however, may not require recovery at a global level after a failure and, therefore, may benefit from local hard failure recovery approaches. For example, parallel stencil computations, which represent a large class of parallel computing applications such as finite-difference methods, exhibit unique computation and communication patterns – multiple iterations, each composed of computation on local data and communication with immediate neighbors. This dissertation shows that, should a multi-process failure of a stencil computation be recovered in a local manner, only the immediate neighbors are immediately affected by the recovery delay of that particular failure while the rest of the domain is allowed to continue the simulation in a failure-agnostic way. This dissertation implements and experimentally evaluates local recovery algorithms as implemented in the FenixLR framework on top of Titan while injecting real failures. Experiments with S3D demonstrate that local recovery provides an optimized resilience solution for stencil-based applications when tolerating node failures occurring as frequently as every 5 seconds on scales up to 250,000+ cores. For example, an overall resilience overhead of less than 14% was observed when injecting node failures every 30 seconds. Furthermore, our experiments confirm the scalability of local recovery stages in FenixLR and demonstrate that process recovery aspects provide better scalability with increasing number of cores than global recovery constructs.

Building on the communication pattern of stencil applications, we model, simulate, and experimentally demonstrate how the delay of recovering from a failure in a local manner

propagates slowly across a machine. Therefore, if a subsequent failure occurs at distant node before the original failure delay has spread to that node, the delay due to the second failure is masked by the delay due to the first one. In general, we show that failure masking allows for the reduction of overhead on total execution time due to recovery from multiple failures down to a level comparable to that of a lower failure count. For example, experimental results have shown an S3D execution where four failures masked each other in an execution with 140,000+ cores to provide an overhead equal to that of a single failure.

The dissertation also describes two application-level optimizations that can be applied to improve the effect and probability of failure masking. These two techniques involve increasing the size of the ghost region and cell-to-rank mapping so that the propagation effect of any node failure is limited or reduced. Both techniques revolve around the concept of decreasing the communication frequency to increase the failure propagation window.

7.2 Future Work

The lines of research presented in this dissertation can be expanded in several directions, including the following:

- **Exploring Data Recovery through Checkpoint-less Data Extrapolation.** Some scientific simulations can tolerate a certain degree of indeterminism in the solution. In other words, instead of requiring an exact solution, some applications may tolerate an approximate one. For example, a PDE simulation that simulates the temperature of a domain might accept some approximation in parts of the domain that are not critical. In the case where ranks of a parallel application simulating non-critical parts are affected by a failure, data that was lost due to the failure can be extrapolated using data that survived in logically neighboring ranks.

  A possible direction for exploring data recovery may be to evaluate the impact of failures when recovered in a checkpoint-less manner as well as when recovered by using checkpoints. The proposed approach can be considered semi-local since, even though the environment would be globally restored, the execution would be resumed locally: survivor ranks are neither required to rollback nor observe the failure – the

framework automatically uses the computational resources of different ranks to fix the environment.

- **Extending On-line Failure Recovery for Multi-Application Workflows.** Scientific simulations may require several applications to be coupled together in order to cooperatively reach a solution. A typical example is the simulation-analysis-visualization workflow in which one of the applications is running a simulation and, in a separate set of nodes, a second application analyzes a particular state of the simulated data to extract conclusions or, for example, to create a visualization for the scientists to manually inspect or monitor application state.

  A possible direction to extend the work in this dissertation is to explore on-line recovery with multi-application workflow. In current scenarios, the failure of one of the applications would stall the workflow as a whole. Specifically, a first step toward this exploration would be to apply global recovery concepts within each application in the workflow and local recovery among applications to recover from failures that affect only one of the applications. A second step could study how simultaneous failures affecting different applications in the workflow can be tolerated. A final study might determine the feasibility of using local recovery within each of the applications as well as between them.

- **Studying Gradual Degradation Feasibility for different Application Types.** The approach for global recovery as well as local recovery presented in this dissertation implements non-shrinking recovery. By pre-allocating spare compute resources, Fenix and FenixLR allow non-shrinking process/node failure recovery which in turn provides the application with the same amount of computational resources (i.e. same number of MPI processes) before and after each failure. This methodology, however, can only tolerate as many failures as spare resources pre-allocated. To overcome this problem altogether, it is expected that future systems will provide dynamic node allocation. For example, the job scheduler of a future system may dynamically provide new resources to jobs that suffered node failures.

Some applications, however, readily support dynamism in the total number of processes and can quickly adapt to shrinking scenarios where resources disappear due to failures. An example can be found in bag-of-tasks applications, where a leader spawns tasks to a set of sub-resources. If a sub-resource is lost, its assigned task can be simply rescheduled to other resources. A possible direction for future work might be to study the feasibility of shrinking recovery for more complex application decompositions, such as bulk-synchrounous applications.

Appendix A

Specification of the Fenix MPI Fault Tolerance library

A.1 Introduction

This appendix provides a specification of Fenix (based on version 1.0.1 [70]), a software library compatible with the Message Passing Interface (MPI) to support fault recovery without application shutdown. The library consists of two modules. The first, termed *process recovery*, restores an application to a consistent state after it has suffered a loss of one or more MPI processes (ranks). The second specifies functions the user can invoke to store application data in Fenix managed redundant storage, and to retrieve it from that storage after process recovery.

A.1.1 Functionality

Fenix is used (1) to repair communicators whose ranks suffered one or more failures detected by the MPI runtime, and (2) to restore state to application variables and arrays from redundant data storage.

Process recovery. Within the Fenix framework, only communicators derived from the communicator returned by `Fenix_Init` are eligible for reconstruction. After communicators have been repaired, they contain the same number of ranks as before the failure occurred, unless the user did not allocate sufficient redundant resources (*spare ranks*) and instructed Fenix not to create new ranks. In this case communicators will still be repaired, but will contain fewer ranks than before the failure occurred.

To ease adoption of MPI fault tolerance, Fenix automatically captures any errors resulting from MPI library calls that experienced a failure due to a damaged communicator (other errors reported by the MPI runtime are ignored by Fenix and are

returned to the application, for handling by the application writer). In other words, programmers do not need to replace calls to the MPI library with calls to Fenix (for example, `Fenix_Send` instead of `MPI_Send`).

Current implementation

Fenix uses MPI's PMPI profiling interface. This currently means that it is incompatible with other software tools that need access to the profiling interface as well. It is expected that this restriction will be lifted soon via MPI extensions similar to that proposed by Schulz and De Supinski [136].

End current implementation

Data recovery. Fenix provides its own redundant data storage API to facilitate data recovery along with process recovery, but the user can choose other data recovery options to meet a variety of application needs. For example, data could be recovered by approximately interpolating values from unaffected, topologically neighboring ranks instead of by reading stored redundant data. In addition, the user may decide to use external libraries such as GVR (Global View Resilience [36]) or SCR (Scalable Checkpoint/Restart [111, 112]) to restore rank data after a failure. The crux is that the program does not have to be shut down and restarted.

Any Fenix function without a return type, e.g. `Fenix_Init`, may be implemented via macros, in which case it cannot be used to resolve function pointers. It is up to the implementation to decide which functions are macros.

Current implementation

Fenix currently does not have a thread safety model.

End current implementation

A.1.2 Terms and format

When describing Fenix functions, we will indicate for each function argument whether it provides an input value (i.e. it is read by the function), an output value (i.e. it is set by the function), or both, using [IN], [OUT], and [INOUT], respectively. If a parameter is

an opaque data type accessed by a handle and the handle itself is not changed by a Fenix function, but the contents of the data type may be, we still label the parameter as [INOUT], in keeping with the MPI specification.

For each function we list whether it is collective or not. If a function is not collective, it has strictly local completion semantics, and other ranks in the associated communicator may safely skip the call. If collective, all ranks in a specific communicator must call that function at logically the same time. Since collective-ness does not imply communication, a collective function may have either local, non-local, or globally synchronizing completion semantics. Collective operations are labeled as follows:

(collective operation, local). If a collective function has local completion semantics, the function will not trigger any synchronization nor communicate across ranks, but results may be incorrect if not called by all ranks in the associated communicator.

(collective operation, non-local). If a collective function has non-local completion semantics, the calling rank potentially communicates with other ranks. All ranks in the associated communicator are allowed to synchronize, but they are not required to; consequently, the function should not be used for synchronization.

(collective operation, global synch). If a collective function is globally synchronizing it is automatically non-local. No ranks in the associated communicator can complete the function until all ranks have called it.

This specification contains sections that give advice to users, or that clarify the current implementation of Fenix. These sections are indicated by

Advice to users
Example advice.
End advice to users

and

Current implementation
Example clarification.
End current implementation,

respectively. They are not part of the specification.

A.2 Initialization, Rank Failure Recovery, and Teardown

A.2.1 Initialization

Fenix Init *(collective operation, global synch)*

```
void Fenix_Init(
        MPI_Comm comm,
        MPI_Comm *newcomm,
        int *role,
        int *argc,
        char ***argv,
        int spare_ranks,
        int spawn,
        MPI_Info info,
        int *error);
```

This function is used (1) to activate the Fenix library, (2) to specify extra resources in case of rank failure, and (3) to create a logical resumption point in case of rank failure.

The program may rely on the state of any variables defined and set before the call to `Fenix_Init`. But note that the code executed before `Fenix_Init` is executed by all ranks in the system (including spare ranks, see below).

`Fenix_Init` is a blocking function call. Specifically, when it is called for the first time (i.e., before any failure has happened), it will block until all ranks in communicator `comm` have reached it. After an error is intercepted by Fenix, and if `Fenix_Init` is chosen as the resumption point (default behavior, which can be modified via the `info` parameter, see below), no ranks are allowed to exit from `Fenix_Init` until all *active ranks* (any ranks in the execution that are not spare ranks are called *active*) have returned control to it.

Current implementation

While `Fenix_Init` is called explicitly only once in a user code, control is transferred into

it indirectly by the Fenix library whenever an active rank calls an MPI function whose associated resilient communicator has been damaged (a condition automatically intercepted by Fenix), based on ULFM's failure notification mechanism. Consequently, the function may experience delay if certain ranks do not call MPI communication functions for a long time, or only call MPI functions involving communicators that were not affected by the error.
End current implementation

Spare ranks are not released from `Fenix_Init` until they have been used by Fenix to repair damaged resilient communicators, or until `Fenix_Finalize` has been called by the active ranks (at which time remaining spare ranks automatically call `MPI_Finalize` and exit). When a failure occurs and is recovered by Fenix, surviving ranks resume execution returning from `Fenix_Init` (or elsewhere depending on the `"resume_mode"` key in `info`). Replacement ranks that are created using `MPI_Comm_spawn` (invoked by Fenix once the spare ranks have been depleted, subject to the rank repair policy specified by the user) start executing the main program, including `MPI_Init` and `Fenix_Init` and any preceding statements. Consequently, spawned replacement ranks experience a different control flow than survivor ranks or spare ranks, which may affect the correctness of MPI calls placed before `Fenix_Init`, especially collective communications. It is the user's responsibility to avoid such problems.

This function must be called by all ranks in `comm`, after `MPI_Init` or `MPI_Init_thread`. All calling ranks must pass the same values for the parameters `comm`, `spare_ranks`, `spawn`, and `info`. `Fenix_Init` must be called exactly once by each rank. It is recommended to access `argc` and `argv` only after executing `Fenix_Init`, since command line arguments passed to this function that apply to Fenix may be removed by `Fenix_Init`.

Parameters:

- `comm` [IN] - communicator that includes any spare ranks (see below) the user deems necessary. It will be used by Fenix to derive a *resilient communicator*. We define a resilient communicator as one whose accesses are monitored by Fenix, which will repair it in case it suffers failed ranks. Any communicator derived from a resilient communicator is automatically resilient itself. To enable successful recovery from failures via Fenix, the user should only use resilient communicators. `MPI_COMM_WORLD`

is a valid value for `comm`. `MPI_COMM_SELF` is not a valid value for `comm`.

- `newcomm` [OUT] - resilient communicator, returned and managed by Fenix and derived from `comm`, to be used by the application instead of `comm`. Ranks in the resilient communicator are assigned in the same order as in `comm`. Let the number of ranks in `comm` be C, the number of spare ranks S (equals `spare_ranks`), and the number of ranks failed thus far F (F is assumed 0 at the first invocation of `Fenix_Init`). Upon exit from `Fenix_Init` `newcomm` contains:

  – $(C - S)$ ranks if `spawn` equals `true`, and

  – $(C - S) - \max(F - S, 0)$ ranks if `spawn` equals `false`.

  If `newcomm` equals `NULL`, Fenix will tacitly replace every occurrence of communicator `comm` in the code with `Fenix_Init`'s resilient output communicator.

  Current implementation

  To fulfill the condition stated in the last sentence, a Fenix implementation may need to use MPI's profiling interface.

  The current implementation uses MPI's profiling interface. If `newcomm` equals `NULL`, the interface will intercept any use of `comm` and replace it with the resilient communicator (which is stored inside Fenix and is not visible to the user).

  End current implementation

  Advice to users

  If `comm` equals `MPI_COMM_WORLD` and `newcomm` equals `NULL`, `MPI_COMM_WORLD` can be used as a resilient communicator. This constitutes a significant convenience, since the user would not need to replace occurrences of `MPI_COMM_WORLD` in the code explicitly with a different resilient communicator.

  End advice to users

- `role` [OUT] - upon return, contains one of the following values, indicating the most recent history of the calling rank:

  - `FENIX_ROLE_INITIAL_RANK` - this is the value returned to all ranks the first time the program is started (i.e. the first time `Fenix_Init` returns after the user invoked a program launcher, e.g. mpirun – not when individual ranks are reconstructed by Fenix to recover from a failure).
  - `FENIX_ROLE_RECOVERED_RANK` - this rank replaces a failed rank since the latest resilient communicator restoration (or initialization) by Fenix. The rank was taken either from the pool of existing spare ranks managed by Fenix, or was newly created by Fenix using `MPI_Comm_spawn`.
  - `FENIX_ROLE_SURVIVOR_RANK` - this rank was not affected by the rank failure that triggered the latest resilient communicator restoration by Fenix.

  The `role` parameter always indicates the role of a rank since the last exit from `Fenix_Init`. For example, assume a certain rank receives the `RECOVERED` role due to a failure. If it survives a subsequent failure, the `role` output parameter will now indicate that this rank is a `SURVIVOR` rank.

- `argc` [INOUT] - pointer to the number of arguments provided by the `argc` argument to `main`, or `NULL`.

- `argv` [INOUT] - pointer to the argument vector provided by the `argv` argument to `main`, or `NULL`.

- `spare_ranks`[1] [IN] - the number of ranks in `comm` that are exempted by Fenix in the construction of the resilient communicator by `Fenix_Init`. These ranks are kept in reserve to substitute for failed ranks. Failed ranks in resilient communicators are replaced by spare or spawned ranks.

  Current implementation

  First, spare ranks are used to substitute failed ranks. When these are depleted, either (1) failed ranks are substituted with spawned ranks (if `spawn` equals `true`), or (2) survivor ranks are compacted to shrink the resilient communicator (if `spawn` equals

[1]While it may be more accurate to use the term spare *MPI processes*, since spare *ranks* taken from `comm` are to be used for substitution in other communicators, we stick with the shorter term for readability.

`false`).

End current implementation

Ranks to be used as spare ranks by Fenix will be available to the application only before `Fenix_Init`, or after they are used to replace a failed rank, in which case they turn into active ranks. This specification refers to the latter as `RECOVERED` ranks.

Note that all spare ranks that have not been used to recover from failures (and, therefore, are still reserved by Fenix and kept inside `Fenix_Init`) will automatically call `MPI_Finalize` and exit when all active ranks have entered the `Fenix_Finalize` call.

- `spawn` [IN] - boolean value used to specify whether Fenix may attempt to spawn replacement ranks or not.

  – If `spawn` equals `false` and insufficient spare ranks are available to replace all failed ranks, no new ranks will be spawned to fill out original communicators. Failures will be resolved by Fenix by "compacting" survivor ranks within their respective resilient communicators, such that they retain the same order as before the failure, but they are numbered successively and contiguously within the shrunk communicator.

    Note that this mode, in combination with requesting no spare ranks, can be used to force a shrinking communicator repair mechanism.

  – If `spawn` equals `true` and insufficient spare ranks are available to replace all failed ranks, some or all required new ranks will be spawned, using `MPI_Comm_spawn`, to fill out original communicators. A correct Fenix implementation must use `MPI_Comm_spawn` to implement this feature, and must pass the entire key-value dictionary of the `info` parameter of `Fenix_Init` in the `info` parameter of `MPI_Comm_spawn`.

    Advice to users

    The last sentence may imply that a user can control where and how to allocate respawned ranks through the `info` parameter, even though that is dependent on

the MPI and Fenix implementations.

End advice to users

Current implementation

Rank spawning in response to a failure is not supported in the current implementation.

End current implementation

- `info` [IN] - `MPI_Info` object to further modify Fenix's process recovery behavior. The application may pass `MPI_INFO_NULL` to indicate default behavior.

  At least the `"resume_mode"` key must be recognized by the Fenix implementation. This key is used to indicate where execution should resume upon rank failure for all active (non-spare) ranks in any resilient communicators, not only for those ranks in communicators that failed. The following values associated with the `"resume_mode"` key must be supported by the Fenix implementation:

  - `"fenix_init"` - execution resumes at logical exit of `Fenix_Init`.

- `error` [OUT] - used to signal that a non-fatal error or special condition was encountered in the execution of `Fenix_Init`, or `FENIX_SUCCESS` otherwise. It has the same value across all ranks released by `Fenix_Init`. If spawning is explicitly disabled (`spawn` equals `false`) and spare ranks have been depleted, Fenix will repair resilient communicators by shrinking them and will report such shrinkage in the `error` return parameter through the value `FENIX_WARNING_SPARE_RANKS_DEPLETED`. If spawning is enabled but fails to create the required new ranks in the absence of a sufficient number of spare ranks, Fenix will repair resilient communicators by shrinking them and will report such shrinkage in the `error` return parameter through the value `FENIX_ERROR_SPAWNING_FAILED`.

Fenix Initialized

```
int Fenix_Initialized(
        int *flag);
```

This function can be used to check whether Fenix has been initialized.

No Fenix functions may be called before `Fenix_Init` or after `Fenix_Finalize`, except `Fenix_Initialized`.

Parameter:

- `flag` [OUT] - true if `Fenix_Init` has been called and false otherwise.

A.2.2 Callback handler function recovery

Fenix Callback register

```
int Fenix_Callback_register(
        void (*recover)(MPI_Comm, int, void*),
        void *callback_data);
```

This function registers a callback to be invoked after a failure has been recovered by Fenix, and right before resuming application execution (e.g., returning from `Fenix_Init` by default). If this function is called more than once, the different callbacks registered will be called in the same order they were registered.

Callbacks will only be invoked by survivor ranks, since spare ranks or respawned ranks had no way to register them before a failure: they only execute code *after* `Fenix_Init` once the Fenix recovery procedure (which includes calling all registered callback functions) is completely finished.

`FENIX_ERROR_CALLBACK_NOT_REGISTERED` will be returned if there is an error while trying to register the callback function.

If a user callback function returns, Fenix will assume that no error occurred within the callback function. Therefore, if an error does occur, it needs to be either solved within the callback or escalated by using mechanisms such as `MPI_Abort` by it.

Parameters:

- `recover` [IN] - the callback function to be registered.
- `callback_data` [IN] - a pointer to application-specific data to be passed as the last parameter when calling the callback. Note that `NULL` is an acceptable value.

Callback functions need to observe the following prototype:

```
void my_recover_callback(
        MPI_Comm comm,
        int error,
        void *callback_data);
```

Since the registration of callback functions is not a collective operation, the callback itself should not perform any MPI communication.

Callback function parameters:

- `comm` [IN] - contains the resilient communicator returned by `Fenix_Init`. When the callback function is invoked by Fenix, this communicator has already been repaired by Fenix and, therefore, callback's `comm` parameter is identical to `newcomm` as returned by `Fenix_Init`.
- `error` [IN] - indicates any error that may have occurred during the recovery process. See Section A.2.1 for more details.
- `callback_data` [IN] - contains the pointer passed when registering the callback (last parameter of `Fenix_Callback_register`). Note that this may be `NULL`.

A.2.3 Querying active ranks

Even though the application can obtain information about the roles of ranks after a failure, that may require a collective communication among ranks in the target resilient communicator. Fenix has access to this information locally and the application can access it by using the following operations.

Fenix Get number of ranks with role

```
int Fenix_Get_number_of_ranks_with_role(
        MPI_Comm comm,
        int role,
        int *number_of_ranks);
```

This function returns the total number of ranks in resilient communicator `comm` that have a particular `role`.

Parameters:

- `comm` [IN] - resilient communicator whose ranks are being queried.
- `role` [IN] - queried role. See the description of the `role` output parameter of `Fenix_Init` for a clarification of the possible values.
- `number_of_ranks` [OUT] - number of ranks in `comm` whose role equals `role`.

Fenix Get role

```
int Fenix_Get_role(
        MPI_Comm comm,
        int rank,
        int *role);
```

This function can be used to query a particular rank in resilient communicator `comm` about its role.

Parameters:

- `comm` [IN] - resilient communicator whose rank is being queried.
- `rank` [IN] - rank whose role is requested.
- `role` [OUT] - the role of the queried rank. See the description of the `role` output parameter of `Fenix_Init` for a clarification of the possible values.

A.2.4 Teardown

Fenix Finalize *(collective operation, global synch)*

```
int Fenix_Finalize(void);
```

This function cleans up all Fenix internal state. If an MPI program using the Fenix library terminates normally (i.e., not due to a call to `MPI_Abort`, or an unrecoverable error) then each such rank must call `Fenix_Finalize` before it exits. It must be called before `MPI_Finalize`, and after `Fenix_Init`. There shall be no Fenix calls after this function, except `Fenix_Initialized`.

As noted in the description of `Fenix_Init`, all spare ranks that have not been used to recover from failures are still reserved by Fenix and kept inside `Fenix_Init`. Fenix will force remaining spare ranks to call `MPI_Finalize` and exit when all active ranks have called `Fenix_Finalize`.

Advice to users

Sometimes users may want to remove ranks proactively from the execution, for example because monitoring data shows that failure of a rank is imminent. This can be accomplished simply by calling `exit` on the targeted ranks, followed by an invocation of `MPI_Barrier`. The removed ranks will not reach the barrier, causing an error among the remaining ranks in the resilient communicator supplied to the barrier function. This error will be intercepted by Fenix, which will attempt to repair the affected communicator, excluding any eliminated ranks[2].

The smaller the communicator used in the invocation of the barrier is chosen, the slower the effect of removing ranks from that communicator may percolate to other ranks.

End advice to users

A.3 Data Storage and Recovery

A.3.1 Overview

Fenix provides options for redundant storage of application data to facilitate application data recovery in a transparent manner. Fenix contains functions to control consistency of collections of such data, as well as their level of persistence. Functions with the prefix `Fenix_Data_` perform store, versioning, restore and other relevant operations and form the Fenix data recovery API.

[2] An out-of-band solution could also be sending a `SIGKILL` signal to the targeted ranks.

The user can select a specific set of application data, identified by its location in memory, label it using `Fenix_Data_member_create`, and copy it into Fenix's redundant storage space through `Fenix_Data_member_(i)store(v)` at a certain point in time.

The user may group semantically similar pieces of data, called members or group members, into disjoint data groups, which can be managed by the `Fenix_Data_group_` set of functions.

Subsequently, `Fenix_Data_commit` assigns a unique time stamp to the resulting data *snapshot*, marking the stored data as potentially recoverable after a loss of ranks. Committing a data group also finalizes all preceding Fenix store operations involving a data group. Therefore, a snapshot is identified by its time stamp (an integer automatically incremented by one) and refers to the memory state of data members in a group. If a member was not stored when creating a particular snapshot S_2, but was stored when creating a previous snapshot S_1, Fenix considers the stored copy in S_1 to be the state of that member in S_2 (see example in Section A.4.4).

Individual data members in a snapshot can be restored whenever they are needed with `Fenix_Data_member_restore`, for example after a failure occurs. We note that the Fenix's data storage and recovery facility aims primarily to support in-memory recovery.

Populating redundant data storage using Fenix may involve dispersion of data created by one rank to other ranks within the system (see, e.g., Chapter 4), making the store operation semantically a collective operation. However, Fenix does not require store operations to be globally synchronizing. For example, execution of `Fenix_Data_member_store` for a particular collection of data could potentially be finished in some ranks, but not yet in others. And if certain ranks nominally participating in the storage operation have no actual data movement responsibility, Fenix is allowed to let them exit the operation immediately. Consequently, Fenix data storage functions should not be used for synchronization purposes.

If snapshots are created by the application (following `Fenix_Init`), both recovered ranks as well as surviving ranks after a failure may be supplied with data from a valid and consistent state taken before the failure occurred. This behavior is controlled by the user.

A.3.2 Managing data storage and recovery constructs

A.3.2.1 Grouping data objects and ranks with *data groups*

Fenix Data group create *(collective operation, local)*

```
int Fenix_Data_group_create(
        int group_id,
        MPI_Comm comm,
        int start_time_stamp,
        int depth);
```

Creates an instantiation of a *Fenix data group*, identified by a unique, user-specified integer.

A Fenix data group provides dual functionality. First, it serves as a container for a set of data objects (*members*) that are committed together, and hence provides transaction semantics. Second, it recognizes that `Fenix_Data_member_store` is an operation carried out collectively by groups of ranks, but not necessarily by all active ranks in the MPI environment. Hence, it adopts the convenient MPI vehicle of *communicators* to indicate the subset of ranks involved.

All ranks in resilient communicator `comm` must pass the same values for all parameters.

Parameters:

- `group_id` [IN] - identifier of the group, unique among all active MPI ranks in the application (not only in `comm`). If a group with this `group_id` was already created in the past and has not been deleted, the `start_time_stamp` and `depth` parameters of this invocation will be ignored, since Fenix automatically determines the correct values based on the previous invocation. The recreated group will logically be the same as the one previously in existence.

  Note that `group_id` functions as a handle to the group, to be used in creating data members associated with the group, storing these members, committing the group, as well as recovering data after a failure. It must be a nonnegative integer less than `FENIX_GROUP_ID_MAX`, with the latter value guaranteed to be at least 2^{30}.

- `comm` [IN] - resilient communicator. The ranks in `comm` participate as a logical unit in the storage and recovery of the data stored by the corresponding `Fenix_Data_member_(i)store(v)` call.

  Current implementation

  If the *buddy rank* mechanism is used for redundant data storage (the default method, see Chapter 4), there have to be at least two ranks in the communicator to be able to recover data after a rank failure. However, if these ranks are collocated on the same processor or within the same node, they are more likely to fail together than if they are located on different nodes. In general, the resilient communicator should be chosen such that it is possible to define a buddy rank that is outside the expected failure envelope of the rank that created the data to be stored.

  End current implementation

- `start_time_stamp` [IN] - each subsequent data snapshot of this group has an index that uniquely identifies the snapshot within the group. This index is called a *time stamp*. The `start_time_stamp` is the index of the first data snapshot of this group to be written, and can be defined by the user (for example, set to zero); this value will be incremented by one automatically each time the group is committed.

  The user-supplied `start_time_stamp` must be a nonnegative integer less than `FENIX_TIME_STAMP_MAX`, with the latter value guaranteed to be at least 2^{30}.

- `depth` [IN] - the number of successive data snapshots (see `Fenix_Data_commit(_barrier)` in Section A.3.4.2) of this group that are retained by Fenix, in addition to the last one, and that can be recovered by calling Fenix data member restore functions (see Section A.3.5). For example, a depth of 0 means Fenix will keep only the necessary data to restore the most recent snapshot, while it will mark older snapshots for deletion. These will be removed automatically whenever `Fenix_Data_commit` or `Fenix_Data_commit_barrier` is called. A depth of -1 means Fenix will not remove any older snapshots automatically. In that case only explicit, manual deletion of out of date snapshots is possible.

The predefined constant `FENIX_DATA_GROUP_WORLD_ID` constitutes a `group_id` identifying a group instance as if created by calling:

```
Fenix_Data_group_create(
        FENIX_DATA_GROUP_WORLD_ID, // group_id
        comm,                      // communicator
        0,                         // start_time_stamp
        0);                        // depth
```

where `comm` is the resilient communicator produced when `Fenix_Init` returned most recently. In other words, `FENIX_DATA_GROUP_WORLD_ID` is a convenient constant to represent a data group involving all active ranks via a reserved `group_id`, an initial time stamp of zero, and a garbage collection depth of zero (i.e., Fenix will keep only the last snapshot).

Applications that do not need the flexibility of the more general Fenix grouping mechanism can, therefore, avoid having to create a specific group and can use this generic group instead.

Fenix Data group delete *(collective operation, local)*

```
int Fenix_Data_group_delete(
        int group_id);
```

Deletes a user-created data group. Any Fenix data group except `FENIX_DATA_GROUP_WORLD_ID` can be deleted.

When a data group is no longer needed, its resources can be released (and its `group_id` be made available for use in other groups) with this function. It will recursively delete all its members. This function may only be called before any store or commit operations have been carried out involving the group or its members (see below).

Parameters:

- `group_id` [IN] - id of the group to be destroyed.

A.3.2.2 Describing application data with *data group members*

Fenix Data member create *(collective operation, local)*

```
int Fenix_Data_member_create(
        int group_id,
        int member_id,
        void *source_buffer,
        int count,
        MPI_Datatype datatype);
```

Fenix data groups are composed of members that describe the actual application data. This function creates a new data member and assigns it into a particular group.

All calling ranks in the group's communicator must pass the same values for the parameters `member_id`, `datatype`, and `group_id`.

Parameters:

- `group_id` [IN] - identifier of the data group containing the member.
- `member_id` [IN] - integer unique within the data group that identifies the data in `source_buffer`. The user-supplied `member_id` must be a nonnegative integer less than `FENIX_MEMBER_ID_MAX`, with the latter value guaranteed to be at least 2^{30}.
- `source_buffer` [IN] - address of data to be copied to redundant storage maintained by Fenix. Note that this parameter may also be specified using the function `Fenix_Data_member_attr_set`. The latter is critical for non-survivor ranks (i.e., `FENIX_ROLE_RECOVERED_RANK`) after a failure. In that case data group members are *implicitly* recreated by Fenix when the programmer calls `Fenix_Data_group_create`, but any pointer to the application data is invalid and must be supplied explicitly by the user for each group member. Survivor ranks will use the source buffer pointer specified before the failure, unless it is overwritten by `Fenix_Data_member_attr_set`.
- `count` [IN] - maximum number of contiguous elements of type `datatype` of the data to be stored[3]. This parameter does not need to be the same in all ranks calling this function.

[3]To avoid problems related to using an `int` to identify sizes (such as 32-bit integers not being big enough to address all the memory, we will use `MPI_Count` once it is adopted by the MPI Forum.

- `datatype` [IN] - data type of each element in `source_buffer`.

Fenix Data member delete *(collective operation, local)*

```
int Fenix_Data_member_delete(
        int group_id,
        int member_id);
```

When a data group member is not needed, it may be deleted using this function.

Note that members can be added to or deleted from a group at any point between the calls to `Fenix_Data_group_create` and `Fenix_Data_group_delete`, but not any longer once any group member has been stored in Fenix' redundant data storage, or once a data group has been committed.

Parameters:

- `group_id` [IN] - identifier of the group containing this member.
- `member_id` [IN] - unique integer within the named group that identifies the data member.

A.3.2.3 Accessing redundancy policies

Fenix Data group get redundancy policy

```
int Fenix_Data_group_get_redundancy_policy(
        int group_id,
        int policy_name,
        void *policy_value,
        int *flag);
```

This function is used to query Fenix for the type of policy it applies to safeguard all meta-data and application data (group members) by dispersing copies of that data. The resilience of data in Fenix' redundant data storage depends on the specified policy.

At least the following policy must be defined:

- `FENIX_DATA_POLICY_PEER_RANK_SEPARATION`, which determines one of the simplest types of data redundancy, namely preserving a copy of the data on a peer rank within the same resilient communicator corresponding to the data group. In this case, the `policy_value` input parameter is the `rank_separation`, and has a default value equivalent to half of the size of the communicator associated with the group. A single copy of the data stored locally on rank `my_rank` will also be stored on rank `(my_rank+rank_separation) mod comm_size`, where `comm_size` equals the size of the communicator associated with the relevant data group. We note that depending on the layout of the ranks of the communicator across the physical resources of the system (nodes, racks, cabinets), different values of the `rank_separation` parameter should be selected to obtain the desired data resilience. For example, assuming a communicator spanning ranks mapped to nodes distributed in two physical cabinets (where ranks 0 to `cabinet_size`-1 are in one cabinet and ranks `cabinet_size` to (2*`cabinet_size`)-1 are in the other), `rank_separation` can be set to `cabinet_size` so that all stored members in the group are replicated in both cabinets.

Parameters:

- `group_id` [IN] - identifier of the group whose policy is sought.
- `policy_name` [IN] - name of policy whose value is sought.
- `policy_value` [OUT] - value of corresponding policy.
- `flag` [OUT] - true if a policy value was extracted; false if no policy is associated with the key.

Fenix Data group set redundancy policy *(collective operation, local)*

```
int Fenix_Data_group_set_redundancy_policy(
int group_id,
int policy_name,
void *policy_value,
int *flag);
```

This function is used to set the type of policy Fenix applies to safeguard all meta-data and application data.

All calling ranks in this group's resilient communicator must pass the same values for the parameters `group_id`, `policy_name`, and the contents of `policy_value`.

Group redundancy policies can only be set before the first store operation of a member of `group_id`, or the first commit operation of the `group_id`. When a member is first stored or the group is first committed, group redundancy is considered frozen and cannot be changed, not even after a failure.

- `group_id` [IN] - identifier of the group.
- `policy_name` [IN] - name of policy.
- `policy_value` [IN] - value of corresponding policy.
- `flag` [OUT] - true if a policy value was set; false if no policy is associated with the key, or if the policy is read-only (this could be a policy that is set at the time Fenix is built or initialized). Upon successful return of this function, all calling ranks are guaranteed to have the same value in the memory position pointed to by `flag`.

A.3.3 Probing and completing asynchronous operations

In many instances programmers can identify useful work to do by the application while a potentially costly Fenix operation is taking place. For this purpose Fenix supports asynchronous operations that return control to the application immediately, but that need to be probed and/or finished later. The functions needed, `Fenix_Data_wait` and `Fenix_Data_test`, are described below.

Fenix Data wait *(collective operation, non-local)*

```
int Fenix_Data_wait(
        Fenix_Request request);
```

Waits for a non-blocking operation identified by `request`.

The user must always call `Fenix_Data_wait` in order to guarantee the successful completion of a non-blocking collective or non-collective operation (unless `Fenix_Data_test` returns with `flag` equaling `true`).

Users are allowed to call `Fenix_Data_wait` with a null or inactive request argument. In this case the operation returns immediately.

Users should be aware that Fenix implementations are allowed, but not required, to synchronize ranks during the completion of a non-local collective operation.

Parameters:

- `request` [IN] - handle to the asynchronous operation.

Fenix Data test

```
int Fenix_Data_test(
        Fenix_Request request,
        int *flag);
```

Tests for the completion of a non-blocking operation identified by `request`.

Users are allowed to call `Fenix_Data_test` with a null or inactive request argument. In this case the operation returns with `flag` equal to `true`.

- `request` [IN] - handle to the asynchronous operation.
- `flag` [OUT] - The call returns immediately with `flag` equal to `true` if the operation is already completed. The call returns with `flag` equal to `false`, otherwise.

A.3.4 Storing and committing application data

A.3.4.1 Storing data group members

Fenix Data member store *(collective operation, non-local)*

```
int Fenix_Data_member_store(
        int group_id,
```

```
int member_id,
Fenix_Data_subset subset_specifier);
```

This function is used to safeguard the data belonging to a particular member of the data group. It places one or more copies of data residing in `source_buffer` (supplied in the call to the function `Fenix_Data_member_create`) in Fenix' redundant data storage.

Current implementation
After creating a copy of this member in the calling rank's memory, Fenix will transfer this local copy to its final destination(s), e.g. non-volatile memory, peer's memory, a file on a local hard disk.
End current implementation

This function may fail if not enough memory can be allocated to store data of the specified size. When the call returns, the application can safely modify the data in `source_buffer` marked for safeguarding, since Fenix is required to have at least one copy of the data member before returning from this function. The saved data, however, will only be available for recovery after being time stamped via committing the group (see Section A.3.4.2). Such recovery requires the group identifier, the member identifier, and the logical time stamp of the saved data.

Multiple calls to `Fenix_Data_member_store` with the same `member_id` without intervening commit calls will lead to storing (parts of) the same application data object. Depending on the value of `subset_specifier`, this may lead to overwriting the data (loss of data), or incremental storage of the full data member.

Parameters:

- `group_id` [IN] - identifier of the group associated with this member.
- `member_id` [IN] - integer label that uniquely identifies a member of the data group (see `Fenix_Data_member_create`). `FENIX_DATA_MEMBER_ALL` will store all members associated with the specified group. All ranks in the group's resilient communicator must use the same value for `member_id`.
- `subset_specifier` [IN] - specifier of the subset of data to be stored. The choice of

this parameter, while in principle strictly local, needs to result in subsets of identical extent in all calling ranks.

Advice to users

The requirement on resultant subset extent minimizes the need for the library to coordinate between the rank whose member needs to be safeguarded and the agent managing Fenix' non-local redundant data storage (which could be another rank in the system), thus resulting in performance improvement. Users are encouraged to use this function instead of `Fenix_Data_member_storev` (see below) whenever possible.

End advice to users

When a `subset_specifier` different than `FENIX_DATA_SUBSET_FULL` is supplied, Fenix will only store the positions in the application source buffer that are in the subset. When `subset_specifier` equals `FENIX_DATA_SUBSET_EMPTY`, no data will be stored.

Fenix Data member storev *(collective operation, non-local)*

```
int Fenix_Data_member_storev(
        int group_id,
        int member_id,
        Fenix_Data_subset subset_specifier);
```

This function is the same as `Fenix_Data_member_store`, except that the extents of the actual subsets realized by the choice of parameter `subset_specifier` and parameter `count` in the call to `Fenix_Data_member_create` can be different in different ranks.

Fenix Data member istore *(collective operation, local)*

```
int Fenix_Data_member_istore(
        int group_id,
        int member_id,
        Fenix_Data_subset subset_specifier,
        Fenix_Request *request);
```

This function has the same effect as `Fenix_Data_member_store`, except that it returns immediately, even before the data has been stored safely. Data in the application source buffer marked for safeguarding may be overwritten once a call to `Fenix_Data_wait` on `request` has returned.

Current implementation

`Fenix_Data_member_istore` copies the application data into local memory before returning and starts the asynchronous transfer to its final destination. Therefore, in the current implementation, marked data in the application source buffer may be overwritten once the call to `Fenix_Data_member_istore` returns.

End current implementation

The result of multiple calls to `Fenix_Data_member_istore` with overlapping subsets and without intervening calls to `Fenix_Data_wait` is undefined.

Parameters:

- `request` [OUT] - handle to the asynchronous store operation.

Fenix Data member istorev *(collective operation, local)*

```
int Fenix_Data_member_istorev(
        int group_id,
        int member_id,
        Fenix_Data_subset subset_specifier,
        Fenix_Request *request);
```

This function is the same as `Fenix_Data_member_istore`, except that the extents of actual subsets realized by the choice of parameter `count` in function `Fenix_Data_member_create` and parameter `subset_specifier` can be different in different ranks.

A.3.4.2 Making stored data recoverable with data group commits

Fenix Data commit *(collective operation, local)*

```
int Fenix_Data_commit(
        int group_id,
        int *time_stamp);
```

This function is used to freeze the current state of a data group, together with all its application data that has been stored in Fenix' redundant storage, and label it with a time stamp, thus creating a snapshot of the stored application data. Only data that has been committed is eligible for recovery through `Fenix_Data_member_restore`.

An application needs to call `Fenix_Data_wait` for all pending asynchronous `Fenix_Data_member_istore(v)` operations in the group before committing.

Note that not all members in the group need to be stored (with `Fenix_Data_member_store` or any other variant) in order for a commit to succeed. See Section A.4.2 for an example of partial storage.

Whenever `Fenix_Data_commit` is called, Fenix will automatically remove any snapshots that are older than the `depth` specified in `Fenix_Data_group_create`.

Because the commit has local completion semantics, it cannot be used for synchronization. Consequently, there is no guarantee that a data member in a snapshot created with `Fenix_Data_commit` is consistent, which is a requirement for being recoverable. A data snapshot is *consistent* with respect to a group member if all ranks in the group's communicator have committed their stores to that group with the same time stamp, and if the member existed on all ranks at the time of the commit. Consistency can be ensured by calling the globally synchronizing function `Fenix_Data_barrier`, see Section A.3.4.4.

Parameters:

- `group_id` [IN] - identifier of the group to commit.
- `time_stamp` [OUT] - pointer to index of the committed data. `NULL` is a valid value, in which case the automatically incremented index is not returned to the application.

  The `time_stamp` parameter will be a nonnegative integer no larger than `FENIX_TIME_STAMP_MAX`, with the latter value guaranteed to be at least 2^{30}.

A.3.4.3 Removing application data

Fenix Data commit delete

The following function removes irretrievably a specific snapshot of a data group. A snapshot is a set of stored group members identified by the time stamp (an integer returned by `Fenix_Data_commit`) they were committed.

This function can be used in addition to, or instead of, the garbage collection that Fenix may perform, which is controlled by the `depth` parameter in `Fenix_Data_group_create`.

```
int Fenix_Data_commit_delete(
        int group_id,
        int time_stamp);
```

- `group_id` [IN] - group whose commit(s) should be removed.
- `time_stamp` [IN] - the time stamp of the requested commit. The special value of `FENIX_DATA_COMMIT_LATEST` will always remove the latest commit. The special value of `FENIX_DATA_COMMIT_ALL` can be used to remove all commits.

A.3.4.4 Data consistency and garbage collection

Fenix Data barrier *(collective operation, global synch)*

```
int Fenix_Data_barrier(
        int group_id);
```

This function enforces consistency of data and meta-data for the data group with label `group_id`. It will remove from redundant storage any inconsistent snapshots, to reduce storage pressure.

- `group_id` [IN] - Fenix data group

Fenix Data commit barrier *(collective operation, global synch)*

```
int Fenix_Data_commit_barrier(
        int group_id,
        int *time_stamp);
```

This function combines, for convenience, the consistency enforcement of `Fenix_Data_barrier` with the time stamp function of `Fenix_Data_commit`. It is equivalent to issuing these two functions, in that order.

Upon completion of this call, only completely consistent snapshots of the specified group remain in Fenix' redundant storage, and any snapshots older than the depth specified in the call to `Fenix_Data_group_create` have been removed (the latter is ignored if depth equals -1).

A.3.5 Recovering application data

After a failure is recovered and control is returned to the application (for example, by returning from `Fenix_Init`), the application may need to restore previous data snapshots. The first step is to recreate the groups using the repaired communicators, which can be done using `Fenix_Data_group_create`, as explained in Section A.3.2.1. Members, however, do not need to be recreated, since both their meta-data (in particular, the `member_id`, the `count`, and the `datatype`) and application data are saved in Fenix' redundant storage.

Fenix Data member restore *(collective operation, global synch)*

```
int Fenix_Data_member_restore(
        int group_id,
        int member_id,
        void *target_buffer,
        int max_count,
        int time_stamp);
```

This function is used to retrieve data from stored and consistently committed members. If the member is inconsistent across the snapshot, it will removed from the snapshot on all ranks, similar to what `Fenix_Data_barrier` would do.

All ranks in the group's resilient communicator must pass the same values for the parameters `group_id`, `member_id`, and `time_stamp`.

This function can only be used if the size of the communicator used to store the data is the same as that at the time of data recovery (this implies non-shrinking communicator recovery in case of a rank loss). See function `Fenix_Data_member_restore_from_rank` for other cases.

If the size of the buffer needing to receive the recovery data is unknown for a particular rank, it can be queried using the functions described in Section A.3.7.3.

Parameters:

- `group_id` [IN] - group that contains the requested data.
- `member_id` [IN] - this value must match the member identifier that was supplied when `Fenix_Data_member_store` was called.
- `target_buffer` [OUT] - the requested stored data will be written contiguously at this local address. If NULL, no attempt will be made to fetch and restore data. This is useful for selective recovery of application data. Each calling rank will receive the selected data from the corresponding rank in the communicator used at the time the snapshot was taken.
- `max_count` [IN] - the requested stored data, if found, will only be recovered if its size is `max_count` times the size of `datatype` or less.
- `time_stamp` [IN] - time stamp of the first snapshot to be inspected for the presence of valid recovery data. Fenix will inspect successively older available consistent snapshot members until it has found for each element of the requested member a valid recovery value. The availability of such data depends on the choice of subsets used in data storage calls, and potentially selective member removal or time stamp skipping. If no value is found, the corresponding element of the receiving buffer is left unchanged. An example of recovery of data from a snapshot taken at a time earlier than that specified in the restore call can be found in Section A.4.4. The special time stamp value of `FENIX_DATA_COMMIT_LATEST` will always identify the group's latest consistent commit.

Fenix Data member restore from rank *(collective operation, global synch)*

```
int Fenix_Data_member_restore_from_rank(
        int group_id,
        int member_id,
        void *target_buffer,
        int max_count,
        int time_stamp,
        int source_rank);
```

This function works the same way as `Fenix_Data_member_restore`, except that the source rank for the data to be recovered is specified explicitly by each calling rank.

Parameters:

- `source_rank` [IN] - specifies the rank (in the resilient communicator associated with `group_id`) that performed the data store and whose data is being recovered. Its value can be set independently by all ranks in the communicator.

We note that this function does not require that the resilient communicator is the same size as the older, failed communicator, and can be used for any recovery pattern consistent with its definition, as long as the value for `source_rank` is valid (within the size of the old communicator).

A.3.6 Managing data subsets

Fenix data group members are used to provide resilient caches for sets of application data that are contiguous in memory. Each set is represented by a pair consisting of {`start_pointer`,`count`}. *Subsets* represent logical subsets of such sets. They allow the user to indicate which elements (zero or more elements between 0 and `count`-1) will be selected for a particular `Fenix_Data_member_store` operation or its variants (see example in Section A.4.3). They provide a convenient mechanism to reduce the burstiness of data traffic to the final destination of stores (such as I/O subsystems) accessed by `Fenix_Data_member_store` calls. They also provide a way to store only the elements of a group member that changed since the last commit call.

When calling `Fenix_Data_member_(i)store(v)` with a non-trivial value of `Fenix_Data_subset`, the subset must properly reference positions contained within the entire data object defined by the value of `count` in the corresponding `Fenix_Data_member_create` call.

The constant `FENIX_DATA_SUBSET_FULL` of type `Fenix_Data_subset` selects all the data indicated by the user via the `count` parameter specified in `Fenix_Data_member_create`. The constant `FENIX_DATA_SUBSET_EMPTY` of type `Fenix_Data_subset` defines a subset containing no elements.

An example of the usage of subsets is as follows. Assume an array of ten elements set initially to a particular set of values. An application iteratively changes the elements in the array, one element per iteration. In this scenario, the application can decide to initially store the entire array, and then, at a specific iteration, store only the changed element by selecting it with subsets.

Another example of an array in a contiguous memory layout is illustrated by Figure A.1. In this example, the second and third `Fenix_Data_member_storev` invocations store subsets of an array by block patterns. Fenix provides a data type to allow users to define the relative location and size of individual blocks.

Current implementation

During the store call and its variants, the Fenix implementation decides how to perform the actual store, based on the data size and granularity of blocks, as well as the properties of underlying I/O subsystems. See `Fenix_Data_member_store` for more details.

End current implementation

Fenix Data subset create

```
int Fenix_Data_subset_create(
        int num_blocks,
        int start_offset,
        int end_offset,
        int stride,
        Fenix_Data_subset *subset_specifier);
```

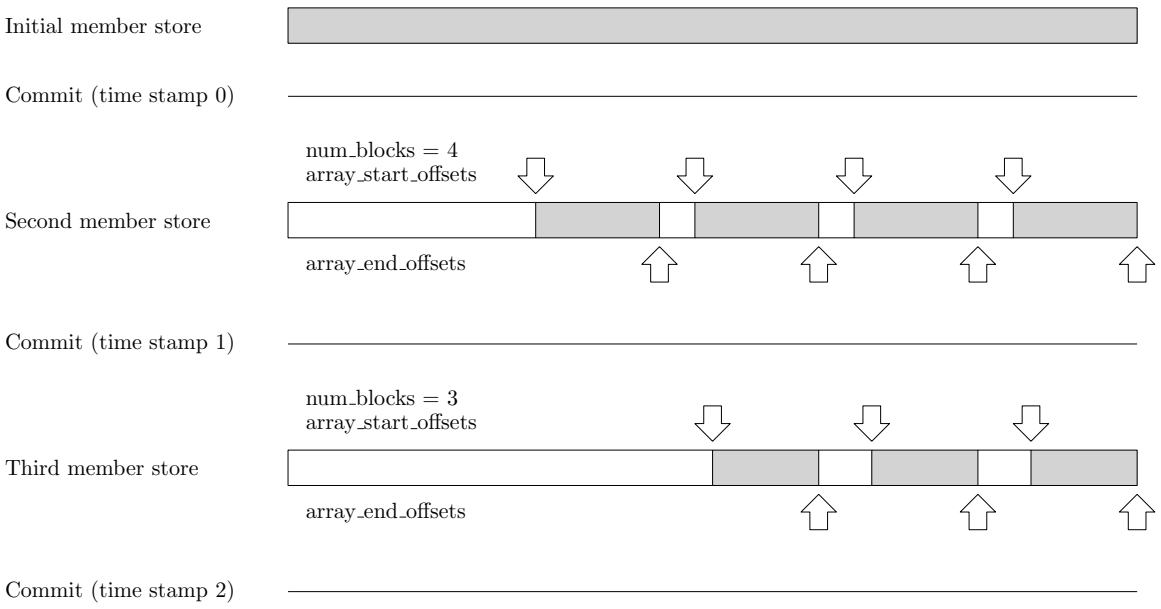


Figure A.1: Incremental member store using *subsets*. Gray areas indicate the data being saved by `Fenix_Data_member_storev` operations.

Creates a subset based on `num_blocks` pairs of {`start_offset,end_offset`}, {`start_offset+stride,end_offset+stride`}, {`start_offset+2*stride,end_offset+2*stride`}, etc. The value of `start_offset` must be smaller than or equal to the value of `end_offset` to indicate non-negative block size. Otherwise, the function returns an error code.

Parameters:

- `num_blocks` [IN] - the number of contiguous data blocks.
- `start_offset` [IN] - an integer indicating the index of the first element of the first data block.
- `end_offset` [IN] - an integer indicating the index of the last element of the first data block.
- `stride` [IN] - regular shift between successive data blocks.
- `subset_specifier` [OUT] - name of the subset specifier, to be used in storing data.

Fenix Data subset createv

```
int Fenix_Data_subset_createv(
```

```
        int num_blocks,
        int* array_start_offsets,
        int* array_end_offsets,
        Fenix_Data_subset *subset_specifier);
```

Creates a subset based on `num_blocks` pairs of {`start_offset,end_offset`}. The value of `start_offset` must be smaller than or equal to `end_offset` to indicate non-negative block size. Otherwise, the function returns an error code.

- `num_blocks` [IN] - the number of contiguous data blocks, which also defines the number of elements in `array_start_offsets` and `array_end_offsets`.
- `array_start_offsets` [IN] - an integer array, which indicates the index of the first elements for each data block (the `start_offset` in the pair {`start_offset,end_offset`}). The value indicates the number of data elements from the beginning of the data registered at `Fenix_Data_member_create`.
- `array_end_offsets` [IN] - an integer array, which indicates the index of the last element for each data block (the `end_offset` in the pair {`start_offset,end_offset`}). The value indicates the number of data elements from the beginning of the data registered at `Fenix_Data_member_create`.
- `subset_specifier` [OUT] - name of the subset specifier, to be used in storing data.

Fenix Data subset delete

```
int Fenix_Data_subset_delete(
        Fenix_Data_subset *subset_specifier);
```

Deletes a previously-created subset. This only refers to meta-data related to the subset; no application data is removed.

- `subset_specifier` [INOUT] - name of the subset specifier, as returned by the `subset_specifier` parameter in `Fenix_Data_subset_create`. The handle is set to `FENIX_SUBSET_NULL`.

A.3.7 Accessing Fenix Data constructs

These functions provide the means to access and alter the information and attributes for Fenix's data recovery and its internals. The status of individual stored objects can be queried by pointing to the corresponding Fenix data group and the `member_id`. Examples in Section A.4.5 and Section A.4.6 show how these functions can be used.

A.3.7.1 Querying data group members

Fenix Data group get number of members

```
int Fenix_Data_group_get_number_of_members(
        int group_id,
        int *number_of_members);
```

Parameters:

- `group_id` [IN] - Fenix data group whose information is sought.
- `number_of_members` [OUT] - number of available distinct member of this group. Manually deleted members are not included in this number.

Fenix Data group get member at position

```
int Fenix_Data_group_get_member_at_position(
        int group_id,
        int *member_id,
        int position);
```

Parameters:

- `member_id` [OUT] - the unique identifier of the `Fenix_Data_member` sought.
- `group_id` [IN] - Fenix data group whose information is sought.

- `position` [IN] - sequence number of the requested `Fenix_Data_member`. `position` must be a value between 0 and `number_of_members`-1 (`number_of_members` as returned by `Fenix_Data_group_get_number_of_members`). The member positions will be returned in the order the user added members to the Fenix data group, i.e. oldest first, newest last (e.g., the first member added by the user will have `position` 0). Deleted members will not be included in this list.

A.3.7.2 Querying commits

Fenix Data group get number of commits

```
int Fenix_Data_group_get_number_of_commits(
        int group_id,
        int *number_of_commits);
```

Parameters:

- `group_id` [IN] - Fenix data group whose information is sought.
- `number_of_commits`[OUT] - number of locally available, distinct commits (snapshots) of this group. This number may include commits that are inconsistent across the group's communicator. Ususally the user will want to know only the number of consistent commits, because no recovery of inconsistent commits can succeed. If there is the possibility of inconsistency, a call to `Fenix_Data_barrier` or other cleanup may be performed first to ensure only consistent commits remain.

Fenix Data group get commit at position

```
int Fenix_Data_group_get_commit_at_position(
        int group_id,
        int position,
        int *time_stamp);
```

Parameters:

- `group_id` [IN] - Fenix data group whose information is sought.
- `position` [IN] - sequence number of the requested commit. `position` must be a value between 0 and `number_of_commits`-1 (`number_of_commits` as returned by `Fenix_Data_group_get_number_of_commits`). Snapshot positions will be returned in the reverse order in which the user committed them, i.e. oldest last, newest first (e.g. the most recent available commit will have `position`=0).
- `time_stamp` [OUT] - the unique index of the commit (snapshot) sought.

A.3.7.3 Accessing data group member attributes

Fenix Data member attr get *(collective operation, non-local)*

```
int Fenix_Data_member_attr_get(
        int group_id,
        int member_id,
        int attribute_name,
        void *attribute_value,
        int *flag,
        int source_rank);
```

Certain properties can be assigned to members of Fenix data groups. These properties, called attributes, can be queried using this function.

All ranks in the group's resilient communicator must pass the same values for the parameters `member_id`, `attribute_name`, and `group_id`.

Parameters:

- `group_id` [IN] - Fenix data group whose information is sought.
- `member_id` [IN] - unique integer within group associated with `group_id` that identifies the data member in Fenix's redundant data storage.

- `attribute_name` [IN] - name of the particular attribute, consisting of the prefix `FENIX_DATA_MEMBER_ATTRIBUTE_`, followed by a suffix. At least the following suffixes must be valid: `SOURCE_BUFFER`, `COUNT`, `DATATYPE`, and `SIZE`.
- `attribute_value` [OUT] - the attribute value of the particular member of the target data group.
- `flag` [OUT] - true if an attribute value was extracted; false if no attribute is associated with the key.
- `source_rank` [IN] - for attributes that are rank-dependent (such as `FENIX_DATA_MEMBER_ATTRIBUTE_COUNT`), specifies the rank in the group's resilient communicator that contains the attribute whose value is sought.

Fenix Data member attr set

```
int Fenix_Data_member_attr_set(
        int group_id,
        int member_id,
        int attribute_name,
        void *attribute_value,
        int *flag);
```

This function can be used to set an attribute related to a member. Attributes can only be set before the first store operation of `member_id` or commit operation of `group_id` that occur after returning from `Fenix_Init`. When a member is stored or a group is committed, attributes are considered frozen until the next failure occurs. After a failure, the execution will be returned from `Fenix_Init`, at which point attributes may be reset before any subsequent stores. In particular, at least the attribute `FENIX_DATA_MEMBER_ATTRIBUTE_SOURCE_BUFFER` must be writable after a failure is recovered.

- `group_id` [IN] - Fenix data group whose information is sought.
- `member_id` [IN] - unique integer within group associated with `group_id` that identifies the data member in Fenix's redundant data storage.

- `attribute_name` [IN] - name of the particular attribute. Attribute names with the suffix `COUNT` and `DATATYPE` are read-only.
- `attribute_value` [IN] - the attribute value of the particular member of the target data group.
- `flag` [OUT] - true if the attribute value was set; false if no attribute is associated with the key or if the attribute is read-only.

A.4 Examples

This section presents some examples on how to use Fenix in different scenarios.

For convenience, the following abbreviations are used:

```
#define BUF          FENIX_DATA_MEMBER_ATTRIBUTE_SOURCE_BUFFER
#define FULL         FENIX_DATA_SUBSET_FULL
#define LATEST       FENIX_DATA_COMMIT_LATEST
#define FENIX_WORLD  FENIX_DATA_GROUP_WORLD_ID
#define COUNT        FENIX_DATA_GROUP_MEMBER_ATTRIBUTE_COUNT
#define TYPE         FENIX_DATA_GROUP_MEMBER_ATTRIBUTE_DATATYPE
#define VP           void *
```

A.4.1 Protecting process and data with Fenix

We show two versions of the same mini-example application, one without fault tolerance, and one augmented with Fenix that tolerates failures in an on-line manner.

```
/* Non-fault-tolerant version */
int main(int argc, char **argv)
{
  int it, A[100], B[50];

  MPI_Init(&argc, &argv);
  initialize(A, B);
```

```
  for(it=0 ; it<1000 ; it++) {
    work1(A, MPI_COMM_WORLD);
    if(A[0] > 200) {
      work2(A, B, MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
}

/* Fault tolerant version with Fenix */
int main(int argc, char **argv)
{
  int it, A[100], B[50], role, flag, error;
  MPI_Comm new_comm_world;

  MPI_Init(&argc, &argv);
  Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
             &argc, &argv,
             10,     // num_spare_ranks
             0,      // no-spawn
             MPI_INFO_NULL,
             &error);
  /* regardless of role, we (re)create the data group */
  Fenix_Data_group_create(
    66,             // group_id
    new_comm_world, // resilient communicator
    0,              // starting index for snapshots
    0);             // depth
  if( !error && role == FENIX_ROLE_INITIAL_RANK) {
    /* no failure occurred */
```

```
    it = 0;
    initialize(A, B);
    Fenix_Data_member_create(66, 90, (VP)&it, 1, MPI_INT);
    Fenix_Data_member_create(66, 91, (VP)A, 100, MPI_INT);
    Fenix_Data_member_create(66, 92, (VP)B,  50, MPI_INT);
    /* make B recoverable */
    Fenix_Data_member_store(66, 92, FULL);
    Fenix_Data_commit_barrier(66, NULL);
  } else if(!error) {
    /* ranks recovered from a failure, now restore data */
    Fenix_Data_member_restore(66, 90, (VP)&it, 1, LATEST);
    Fenix_Data_member_restore(66, 91, (VP)A, 100, LATEST);
    Fenix_Data_member_restore(66, 92, (VP)B,  50, LATEST);
    /* need to say where member data lives, for future stores */
    Fenix_Data_member_attr_set(66, 90, BUF, (VP)&it, &flag);
    Fenix_Data_member_attr_set(66, 91, BUF, (VP)A,   &flag);
    Fenix_Data_member_attr_set(66, 92, BUF, (VP)B,   &flag);
  } else {
    // There was an error in Fenix
    MPI_Abort(MPI_COMM_WORLD, -1);
  }

  /* it may be initialized or recovered */
  for( ; it<1000; it++ ) {
    Fenix_Data_member_store(66, 90, FULL);
    work1(A, new_comm_world);
    if(A[0] > 200) {
      work2(A, B, new_comm_world);
      Fenix_Data_member_store(66, 92, FULL);
    }
```

```
    Fenix_Data_member_store(66, 91, FULL);
    Fenix_Data_commit_barrier(66, NULL);
  }
  Fenix_Finalize();
  MPI_Finalize();
}
```

A.4.2 Storing select members of a data group

The following scenario is a valid Fenix usage demonstrating that not all members need to be stored before committing a group. It assumes a data group labeled 66 was previously created.

```
// Create members
Fenix_Data_member_create(66, 0, (void *)&a, 1, MPI_INT);
Fenix_Data_member_create(66, 1, (void *)&b, 1, MPI_INT);
// Store members as part of commit with time stamp 0
a = myrank;
b = myrank+1;
Fenix_Data_member_store(66, 0, FENIX_DATA_SUBSET_FULL);
Fenix_Data_member_store(66, 1, FENIX_DATA_SUBSET_FULL);
Fenix_Data_commit(66, &ts); // after this, ts equals 0
// Store only member 'b' for commit with time stamp 1
b = myrank+100;
Fenix_Data_member_store(66, 1, FENIX_DATA_SUBSET_FULL);
Fenix_Data_commit(66, &ts); // after this, ts equals 1
```

A.4.3 Storing data objects with subsets

```
/* Non-fault-tolerant version */
int main(int argc, char **argv)
{
```

```
  int it;
  double A[10000];
  const int lda = 100;

  MPI_Init(&argc, &argv);
  initialize(A);

  for(it=0 ; it<100 ; it++) {
    work1(A[lda*it + it], MPI_COMM_WORLD);
  }
}

/* Fault tolerant version with Fenix */
int main(int argc, char **argv)
{
  int it, A[10000], offsets[100], sizes[100], role;
  int start_offset_A[100], end_offset_A[100];
  const int lda = 100;
  Fenix_Data_subset subset_LU;

  MPI_Init(&argc, &argv);
  Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role, ...);
  /* regardless of role, we (re)create the data group */
  Fenix_Data_group_create(
    66,             // group_id
    new_comm_world, // resilient communicator
    0,              // starting index for snapshots
    0);             // depth
  if(!error && role == FENIX_ROLE_INITIAL_RANK) {
    /* no failure occurred */
    it = 0;
```

```
    initialize(A);
    Fenix_Data_member_create(66, 90, (VP)&it, 1,     MPI_INT);
    Fenix_Data_member_create(66, 91, (VP)A,   10000, MPI_DOUBLE);
    Fenix_Data_member_store(66, FENIX_DATA_MEMBER_ALL, FULL);
    Fenix_Data_commit(66, NULL);
  } else {
    /* ranks recovered from a failure, now restore data */
    Fenix_Data_restore(66, 90, (VP)&it, 1,     LATEST);
    Fenix_Data_restore(66, 91, (VP)A,   10000, LATEST);
  }

  for( ; it<100 ; it++) {
    Fenix_Data_member_store(66, 90, FULL);
    /* Create a subset */
    for( j = it; j < 100; j++ ) {
       start_offset_A[j] = j*100 + j;
       end_offset_A[j] =  start_offset_A[j] + lda;
    }
    Fenix_Data_subset_createv(100-it, start_offset_A,
                              end_offset_A, &subset_LU);

    work1(A[lda*it + it]);
    Fenix_Data_member_store(66, 91, subset_LU);

    Fenix_Data_commit(66, NULL);
    Fenix_Data_subset_delete(&subset_LU); // garbage collection
  }
}
```

A.4.4 Recovering data from older time stamps

An example of restoring members not included in the snapshot with the specified time stamp, but present in an earlier snapshot, can seen in lines 20–23 of the following scenario. The data group is labeled 66.

```
// Create members
Fenix_Data_member_create(66, 0, (void*)&a, 1, MPI_INT);
Fenix_Data_member_create(66, 1, (void*)&b, 1, MPI_INT);
// Store members for snapshot with time stamp 0
a = myrank;
b = myrank+1;
Fenix_Data_member_store(66, 0, FENIX_DATA_SUBSET_FULL);
Fenix_Data_member_store(66, 1, FENIX_DATA_SUBSET_FULL);
Fenix_Data_commit(66, &ts); // after this, ts=0
// Store member 'b' for snapshot with time stamp 1
b = myrank+100;
Fenix_Data_member_store(66, 1, FENIX_DATA_SUBSET_FULL);
Fenix_Data_commit(66, &ts); // after this, ts=1
// Store member 'a' for snapshot with time stamp 2
a = myrank+200;
Fenix_Data_member_store(66, 0, FENIX_DATA_SUBSET_FULL);
Fenix_Data_commit(66, &ts); // after this, ts=2


// Restore members
Fenix_Data_member_restore(66, 0, (void*)&new_a, 1, 1);
// new_a now contains "myrank" (line 5)
Fenix_Data_member_restore(66, 1, (void*)&new_b, 1, 1);
// new_b now contains "myrank+100" (line 11)
```

A.4.5 Recovering one member of a data group

This example assumes that ranks have knowledge of (1) the group identifier `group_id`, (2) the size of the communicator associated with that group (same size as `new_comm_world`), (3) the features of the member sought (in particular, `member_id`, `count`, and `datatype`) and (4) the specific time stamp `ts` of the sought consistent snapshot.

```
Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
           &argc, &argv,
           num_spare_ranks,
           0, // no-spawn
           MPI_INFO_NULL,
           &error);
if( !error && role != FENIX_ROLE_INITIAL_RANK ) {
  // Failure successfully recovered
  Fenix_Data_group_create(group_id, new_comm_world,
           0,  // These last two params are ignored,
           0); // since group_id already existed
  MPI_Type_size(datatype, &dt_size);
  uint8_t recovered_data = (uint8_t *) malloc(count*dt_size);
  Fenix_Data_member_restore(
           group_id, member_id, (VP)&recovered_data, count, ts);
  // At this point, the application has its recovered data in
  // all positions of member_pointers.
  // Now, the application should inspect these elements to try
  // and determine what to do with the recovered data.
}
```

A.4.6 Recovering all members of a data group

This example assumes that ranks have the knowledge of (1) the group identifier `group_id` as well as (2) the size of the communicator associated with that group (same size as

new_comm_world).

This example assumes that the recovered rank has no knowledge about the application data contained in the members that were stored. This is a corner case, since the application should be aware of the data associated with a member identifier in a group.

```
Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
           &argc, &argv,
           num_spare_ranks,
           0, // No-spawn
           MPI_INFO_NULL,
           &error);
MPI_Comm_rank(new_comm_world, &my_rank);
if( !error && role != FENIX_ROLE_INITIAL_RANK ) {
  // Failure successfully recovered
  Fenix_Data_group_create(group_id, new_comm_world,
           0,  // These last two params are ignored,
           0); // since group_id already existed

  Fenix_Data_group_get_number_of_members(
           group_id, &number_of_members);

  uint8_t **member_pointers = (uint8_t **)
           malloc(number_of_members*sizeof(uint8_t *));
  int *member_counts = (int *)
           malloc(number_of_members*sizeof(int));
  MPI_Datatype *member_datatypes = (MPI_Datatype *)
           malloc(number_of_members*sizeof(MPI_Datatype));

  for(int m=0 ; m<number_of_members ; m++) {
    Fenix_Data_group_get_member_at_position(
           group_id, &member_id, m);
```

```
        Fenix_Data_member_attr_get(group_id, member_id, COUNT,
                (VP)&member_counts[m], &flag, my_rank);
        Fenix_Data_member_attr_get(group_id, member_id, TYPE,
                (VP)&member_datatypes[m], &flag, my_rank);
        MPI_Type_size(member_datatypes[m], &dt_size);
        member_pointers[m] = (uint8_t *) malloc(count*dt_size);
        Fenix_Data_group_get_commit_at_position(
                group_id, 0, &time_stamp);
        Fenix_Data_member_restore(
                group_id, member_id, (VP)&(member_pointers[m]),
                member_counts[m], time_stamp);
    }
    // At this point, the application has its recovered data in
    // all positions of member_pointers.
    // Now, the application should inspect these elements to try
    // and determine what to do with the recovered data.
}
```

A.4.7 Changing attributes of a data group member

This example demonstrates how to use `Fenix_Data_member_attr_set()` to change attributes before the first `Fenix_Data_member_store` call.

```
int ival;
double dval[2];
int new_count = 2;
MPI_Datatype member_datatype = MPI_DOUBLE;

Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
            &argc, &argv,  num_spare_ranks,
            0, // No-spawn
```

```
            MPI_INFO_NULL, &error);

MPI_Comm_rank(new_comm_world, &my_rank);

Fenix_Data_group_create(66, new_comm_world, 0, 0);

// Fenix Data is created for single integer
Fenix_Data_member_create(66, 90, (VP)&ival, 1, MPI_INT);

// Change the address of the source buffer
Fenix_Data_member_attr_set(66, 90, BUF, (VP)dval,  &flag, my_rank);
// Change the number of elements
Fenix_Data_member_attr_set(66, 90, COUNT,
            (VP) &new_count, &flag, my_rank);
// Change the datatype
Fenix_Data_member_attr_set(66, 90, TYPE,
            (VP)&member_datatype, &flag, my_rank);

// First store call
Fenix_Data_member_store(66, 90, FULL);
```

The following example shows how to change an attribute to point Fenix to a new source buffer for data recovery after a failure has occurred.

```
Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
            &argc, &argv,
            num_spare_ranks,
            0, // No-spawn
            MPI_INFO_NULL,
            &error);

// (re-)create group, regardless of role of calling rank
```

```
Fenix_Data_group_create(66, new_comm_world,
            0,  // starting index for snapshots; ignored if
                // not INITIAL_RANK
            0); // depth; ignored if not INITIAL_RANK

// Only create the data member once
if( !error && role == FENIX_ROLE_INITIAL_RANK) {
  // allocate space for the member
  double *list = (double *) malloc(sizeof(double)*1000);
  Fenix_Data_member_create(66, 91, (VP)list, 1000, MPI_DOUBLE);
}

// recovered ranks need to reallocate space
if( !error && role == FENIX_ROLE_RECOVERED_RANK) {
  double *list = (double *) malloc(sizeof(double)*1000);
  Fenix_Data_member_attr_set(66, 91, BUF, (VP)list, &flag);
}

// At this point, the application can issue store calls for
// the data member associated with list
```

References

[1] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. In *IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, pages 83–87, Aug 2010.

[2] L. Alvisi, E. Elnozahy, S. Rao, S. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing. Digest of Papers*, FTCS 1999 - DSN, pages 242–249, 1999.

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

[4] G. Aupy, A. Benoit, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. On the Combination of Silent Error Detection and Checkpointing. In *The 19th IEEE Pacific Rim International Symposium on Dependable Computing - 2013*, PRDC 2013, Vancouver, Canada, Dec 2013. IEEE.

[5] R. F. Barrett, S. D. Hammond, C. T. Vaughan, D. W. Doerfler, M. A. Heroux, J. P. Luitjens, and D. Roweth. Navigating an evolutionary fast path to exascale. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 355–365, Washington, DC, USA, 2012. IEEE Computer Society.

[6] 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.

[7] 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.

[8] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining In-situ and In-transit Processing to Enable Extreme-scale Scientific Analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC 2012, pages 49:1–49:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[9] 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.

[10] 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.

[11] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of user-level failure mitigation support in MPI. In *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, EuroMPI 2012, pages 193–203, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. A Checkpoint-on-Failure Protocol for Algorithm-Based Recovery in Standard MPI. In C. Kaklamanis, T. Papatheodorou, and P. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 477–488. Springer Berlin Heidelberg, 2012.

[13] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Extending the scope of the Checkpoint-on-Failure protocol for forward recovery in standard MPI. *Concurrency and Computation: Practice and Experience*, 2013.

[14] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified Model for Assessing Checkpointing Protocols at Extreme-Scale. Rapport de recherche RR-7950, INRIA, Oct 2012.

[15] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 33:1–33:11, New York, NY, USA, 2011. ACM.

[16] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications*, 2013.

[17] M. S. Bouguerra, A. Gainaru, L. B. Gomez, F. Cappello, S. Matsuoka, and N. Maruyam. Improving the Computing Efficiency of HPC Systems Using a Combination of Proactive and Preventive Checkpointing. *International Parallel and Distributed Processing Symposium*, 0:501–512, 2013.

[18] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[19] A. Bouteiller, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, and Y. Robert. Multi-criteria Checkpointing Strategies: Response-Time versus Resource Utilization. In *Euro-Par*, Euro-Par 2013, pages 420–431, 2013.

[20] 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.

[21] 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.

[22] 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*, pages 242–250, 2003.

[23] P. J. Braam. Lustre: A Scalable, High Performance File System. `https://http://www.lustre.org/docs.html`, 2004. Accessed: 2014-03-15.

[24] B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80:372–380, 2013.

[25] J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Wiley-Interscience, New York, NY, USA, 1987.

[26] J. Cao, K. Arya, and G. Cooperman. Transparent Checkpoint-Restart over InfiniBand. *CoRR*, abs/1312.3938, 2013.

[27] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov 2009.

[28] K. Carlberg, C. Bou-Mosleh, and C. Farhat. Efficient non-linear model reduction via a least-squares Petrov–Galerkin projection and compressive tensor approximations. *International Journal for Numerical Methods in Engineering*, 86(2):155–181, April 2011.

[29] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Combining Process Replication and Checkpointing for Resilience on Exascale Systems. Rapport de recherche RR-7951, INRIA, May 2012.

[30] R. Castain, J. Ladd, D. Solt, and G. Brown. PMIx Community Meeting at SC'15, Austin, TX, USA, November 2015.

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

[32] Y. S. Chang, S. Y. Cho, and B. Y. Kim. Performance evaluation of the striped checkpointing algorithm on the distributed RAID for cluster computer. In *Proceedings of the 2003 international conference on Computational science: PartII*, ICCS 2003, pages 955–962, Berlin, Heidelberg, 2003. Springer-Verlag.

[33] 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.

[34] Z. Chen. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP 2013, pages 167–176, New York, NY, USA, 2013. ACM.

[35] Z. Chen and J. Dongarra. Algorithm-Based Fault Tolerance for Fail-Stop Failures. *IEEE Trans. Parallel Distrib. Syst.*, 19(12):1628–1641, Dec 2008.

[36] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber, et al. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *Journal of Computational Science*, 2015.

[37] Y.-H. Choi and M. Malek. A fault-tolerant systolic sorter. *IEEE Transactions on Computers*, 37(5):621–624, 1988.

[38] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC 2012, pages 58:1–58:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[39] 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.

[40] A. Cunei and J. Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE 2006, pages 68–77, New York, NY, USA, 2006. ACM.

[41] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 2006.

[42] 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.*, 51(1):129–159, Feb. 2009.

[43] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[44] T. Davies and Z. Chen. Correcting soft errors online in LU factorization. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC 2013, pages 167–178, New York, NY, USA, 2013. ACM.

[45] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello. Optimization of Cloud Task Processing with Checkpoint-restart Mechanism. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2013, pages 64:1–64:12, New York, NY, USA, 2013. ACM.

[46] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu. Noncollective Communicator Creation in MPI. In Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, editors, *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 282–291. Springer, 2011.

[47] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen. Matrix Multiplication on GPUs with On-Line Fault Tolerance. In *IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2011*, pages 311–317, 2011.

[48] J. Dongarra and e. a. . The International Exascale Software Project: a Call To Cooperative Action By the Global High-Performance Community. *Int. J. High Perform. Comput. Appl.*, 23(4):309–322, Nov. 2009.

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

[50] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP 2012, pages 225–234, New York, NY, USA, 2012. ACM.

[51] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, ScalA 2011, pages 11–14, New York, NY, USA, 2011. ACM.

[52] S. El Sayed, S. Graf, M. Hennecke, D. Pleiter, G. Schwarz, H. Schick, and M. Stephan. *Using GPFS to Manage NVRAM-Based Storage Cache*, pages 435–446. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[53] E. Elnozahy and W. Zwaenepoel. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, 1992.

[54] 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.

[55] C. Engelmann and S. Bohm. Redundant Execution of HPC Applications with MR-MPI. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks*, PDCN 2011, pages 31–38, Innsbruck, Austria, 2011. ACTA Press, Calgary, AB, Canada.

[56] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365–376, June 2011.

[57] ExaCT Center for Exascale Simulation of Combustion in Turbulence. Performance Modeling for ExaCT Codes, internal presentation, September 2012.

[58] G. E. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[59] C. Feichtinger, S. Donath, H. Kstler, J. Gtz, and U. Rde. Walberla: {HPC} software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105 – 112, 2011. Simulation Software for Supercomputers.

[60] K. Ferreira, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, R. Brightwell, and R. Riesen. rMPI: Increasing Fault Resiliency in a Message-Passing Environment.

Technical Report SAND2011-2488, Sandia National Laboratories, Albuquerque, NM, April 2011.

[61] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, R. Brightwell, and T. Kordenbrock. Increasing Fault Resiliency in a Message-Passing Environment, Oct 2009.

[62] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2011, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[63] K. B. Ferreira, S. Levy, P. Widener, D. Arnold, and T. Hoefler. Understanding the effects of communication and coordination on checkpointing at scale. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC14)*, 2014.

[64] 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.

[65] M. P. Forum. Mpi: A message-passing interface standard. Technical report, University of Tennessee Knoxville, Knoxville, TN, USA, 1994.

[66] 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 IEEE. Some passages of this dissertation have been reprinted from this publication with permission from the copyright holder, 2014.

[67] 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 *The 24th International ACM Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15. ©2015 ACM. Some passages of this dissertation have been reprinted from this publication with permission from the copyright holder, 2015.

[68] M. Gamell, K. Teranishi, M. A. 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*, SC '15, pages 70:1–70:12. ©2015 ACM. Some passages of this dissertation have been reprinted from this publication with permission from the copyright holder, 2015.

[69] M. Gamell, K. Teranishi, H. Kolla, J. Mayo, M. A. Heroux, J. Chen, and M. Parashar. Scalable Failure Masking for Stencil Computations using Ghost Region Expansion and Cell to Rank Re-mapping (accepted for publication). *SIAM Journal on Scientific Computing*, 2017.

[70] M. Gamell, R. F. Van der Wijngaart, K. Teranishi, and M. Parashar. Specification of Fenix MPI Fault Tolerance library, version 1.0.1. Technical Report SAND2016-10522, Sandia National Laboratories, Livermore, CA, October 2016.

[71] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer.

[72] 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.

[73] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications. In *IEEE 26th International Parallel and Distributed Processing Symposium, 2012*, IPDPS 2012, pages 1216–1227, 2012.

[74] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, and R. A. van de Geijn. Fault-Tolerant High-Performance Matrix Multiplication: theory and practice. In *International Conference on Dependable Systems and Networks. DSN 2001.*, pages 47–56, 2001.

[75] 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*, DSN '15, pages 37–44, Washington, DC, USA, 2015. IEEE Computer Society.

[76] D. Hakkarinen and Z. Chen. Algorithmic Cholesky factorization fault recovery. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–10, 2010.

[77] 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.

[78] 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*, SC '15, pages 31:1–31:12, New York, NY, USA, 2015. ACM.

[79] 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.

[80] M. A. Heroux. Resilience research is essential but failure is unlikely, April 2016. SIAM PP16, Resilience Toward Exascale Computing Symposium.

[81] K.-H. Huang and J. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

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

[83] J. Hursey and R. Graham. Building a Fault Tolerant MPI Application: A Ring Communication Example. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, IPDPSW, pages 1549–1556, 2011.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] I. Jangjaimon and N.-F. Tzeng. Adaptive Incremental Checkpointing via Delta Compression for Networked Multicore Systems. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 7–18, May 2013.

[89] Y. Jia, G. Bosilca, P. Luszczek, and J. J. Dongarra. Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 88. ACM, 2013.

[90] C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, M. Wolf, W. Liao, A. Choudhary, et al. Adaptive io system (adios). *Cray User's Group*, 2008.

[91] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi. Using Cross-layer Adaptations for Dynamic Data Management in Large Scale Coupled Scientific Workflows. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC 2013, pages 74:1–74:12, New York, NY, USA, 2013. ACM.

[92] 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*, PODC 1988, pages 171–181, New York, NY, USA, 1988. ACM.

[93] 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.

[94] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 51–60, New York, NY, USA, 2006. ACM.

[95] 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.

[96] D. S. Katz, J. Daly, N. DeBardeleben, M. Elnozahy, B. Kramer, L. Lathrop, N. Nystrom, K. Milfeld, S. Sanielevici, S. Cott, and L. Votta. 2009 fault tolerance for extreme-scale computing workshop, Albuquerque, NM - March 19-20, 2009. Technical Report ANL/MCS-TM-312, Argonne National Laboratory, December 2009.

[97] R. J. Kee, F. M. Rupley, J. A. Miller, M. E. Coltrin, J. F. Grcar, E. Meeks, H. K. Moffat, A. E. Lutz, G. DixonLewis, M. D. Smooke, J. Warnatz, G. H. Evans, R. S. Larson, R. E. Mitchell, L. R. Petzold, W. C. Reynolds, M. Caracotsios, W. E. Stewart, P. Glarborg, C. Wang, and O. Adigun. Chemkin collection. Reaction Design, Inc., San Diego, CA., 2000.

[98] K. Kharbas, D. Fiala, F. Mueller, K. B. Ferreira, and C. Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *ICDCS*, pages 615–626. IEEE, 2012.

[99] F. B. Kjolstad and M. Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 4:1–4:9, New York, NY, USA, 2010. ACM.

[100] O. R. N. Laboratory. Atlas Transition. `https://www.olcf.ornl.gov/kb_articles/atlas-transition/`, 2014. Accessed: 2014-03-15.

[101] C. Leopold and M. Süß. Observations on MPI-2 support for hybrid master/slave applications in dynamic and heterogeneous environments. In *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, EuroPVM/MPI 2006, pages 285–292, Berlin, Heidelberg, 2006. Springer-Verlag.

[102] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[103] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[104] R.-S. Liu, C.-L. Yang, C.-H. Li, and G.-Y. Chen. Duracache: A durable ssd cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 166:1–166:6, New York, NY, USA, 2013. ACM.

[105] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Păun, and S. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *IEEE International Symposium on Parallel and Distributed Processing*, IPDPS 2008, pages 1–9, April 2008.

[106] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[107] G. A. McMECHAN. Migration by extrapolation of time-dependent boundary values*. *Geophysical Prospecting*, 31(3):413–420, 1983.

[108] E. Meneses, X. Ni, T. Jones, and D. Maxwell. Analyzing the interplay of failures and workload on a leadership-class supercomputer. In *Cray User Group 2015, Chicago, IL*, 2015.

[109] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[110] B. Mills, R. E. Grant, K. B. Ferreira, and R. Riesen. Evaluating Energy Savings for Checkpoint/Restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, E2SC 2013, pages 6:1–6:8, New York, NY, USA, 2013. ACM.

[111] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski. Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system. *Lawrence Livermore National Laboratory (LLNL), Tech. Rep. LLNL-TR-440491*, 2010.

[112] 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.

[113] R. Netzer and Y. Xu. Replaying distributed programs without message logging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 137–147, 1997.

[114] X. Ni, E. Meneses, N. Jain, and L. V. Kalé. ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2013, pages 7:1–7:12, New York, NY, USA, 2013. ACM.

[115] X. Ni, E. Meneses, and L. Kale. Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 364–372, 2012.

[116] B. Nicolae. Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS 2013, pages 19–28, Washington, DC, USA, 2013. IEEE Computer Society.

[117] B. Nicolae and F. Cappello. BlobCR: Efficient Checkpoint-restart for HPC Applications on IaaS Clouds Using Virtual Disk Image Snapshots. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2011, pages 34:1–34:12, New York, NY, USA, 2011. ACM.

[118] B. Nicolae and F. Cappello. AI-Ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC 2013, pages 155–166, New York, NY, USA, 2013. ACM.

[119] B. Obama. Executive Order – Creating a National Strategic Computing Initiative. `https://www.whitehouse.gov/the-press-office/2015/07/29/executive-order-creating-national-strategic-computing-initiative`, July 2015. Accessed: 2016-05-17.

[120] 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.

[121] 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.

[122] J. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on*, pages 76–85, 1996.

[123] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986, 1998.

[124] F. Quaglia. Combining periodic and probabilistic checkpointing in optimistic simulation. In *Parallel and Distributed Simulation, 1999. Proceedings. Thirteenth Workshop on*, pages 109–116, 1999.

[125] 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.

[126] R. Rajachandrasekar, X. Ouyang, X. Besseron, V. Meshram, and D. K. Panda. Can Checkpoint/Restart Mechanisms Benefit from Hierarchical Data Staging? In *Proceedings of the International Conference on Parallel Processing - Volume 2*, Euro-Par 2011, pages 312–321, Berlin, Heidelberg, 2012. Springer-Verlag.

[127] I. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[128] C. A. Reiss, J. Lofstead, and R. Oldfield. Implementation and evaluation of a staging proxy for checkpoint i/o. *CSRI Summer Proceedings 2008*, page 131, 2008.

[129] R. Riesen, K. Ferreira, D. Da Silva, P. Lemarinier, D. Arnold, and P. G. Bridges. Alleviating scalability issues of checkpointing protocols. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC 2012, pages 18:1–18:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[130] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello. On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications. In *Euro-Par (1)*, Euro-Par 2011, pages 567–578, 2011.

[131] 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.

[132] G. Roth, G. Roth, J. Mellor-crummey, J. Mellor-crummey, K. Kennedy, K. Kennedy, R. G. Brickner, and R. G. Brickner. Compiling stencils in high performance fortran. In *In Supercomputing 97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM*, pages 1–20. ACM Press, 1997.

[133] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and Modeling of a Non-blocking Checkpointing System. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC 2012, pages 19:1–19:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[134] A. Schäfer and D. Fey. Libgeodecomp: A grid-enabled library for geometric decomposition codes. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 285–294, Berlin, Heidelberg, 2008. Springer-Verlag.

[135] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[136] M. Schulz and B. R. De Supinski. PN MPI tools: A whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 30. ACM, 2007.

[137] P. Shastry and K. Venkatesh. Selection of a Checkpoint Interval in Coordinated Checkpointing Protocol for Fault Tolerant Open MPI. *International Journal on Computer Science and Engineering*, 2(6):2064–2070, 2010.

[138] S. Sinha and M. Parashar. Adaptive runtime partitioning of amr applications on heterogeneous clusters. In *Proceedings of the 3rd IEEE International Conference on Cluster Computing*, volume 22, pages 435–442. IEEE Computer Society, 2001.

[139] 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.

[140] 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*, ExaMPI '14, pages 9–19, Piscataway, NJ, USA, 2014. IEEE Press.

[141] R. Subramaniyan, V. Aggarwal, A. Jacobs, and A. George. FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems. In H. R. Arabnia, editor, *14th Annual European Symposium on Algorithms*, ESA 2006, pages 3–9. CSREA Press, 2006.

[142] Q. Sun, T. Jin, M. Romanus, H. Bui, F. Zhang, H. Yu, H. Kolla, S. Klasky, J. Chen, and M. Parashar. Adaptive data placement for staging-based coupled scientific workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 65:1–65:12, New York, NY, USA, 2015. ACM.

[143] W. W. Symes. Reverse time migration with optimal checkpointing. *Geophysics*, pages 213–221, 2007.

[144] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 408–413, 2009.

[145] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Traff. MPI at Exascale. *Procceedings of SciDAC*, 2, 2010.

[146] 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*, pages 25–36, June 2014.

[147] Top500.org. The Top500 List. `http://www.top500.org/list/2016/11/`, 2016. Accessed: 2016-12-15.

[148] 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.

[149] S. S. Vadhiyar and J. Dongarra. SRS: A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[150] B. Vinnakota and N. Jha. A dependence graph-based approach to the design of algorithm-based fault tolerant systems. In *20th International Symposium on Fault-Tolerant Computing. FTCS-20. Digest of Papers.*, pages 122–129, 1990.

[151] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration in HPC environments. In *International Conference for High Performance Computing, Networking, Storage and Analysis.*, pages 1–12, 2008.

[152] S.-J. Wang and N. Jha. Algorithm-based fault tolerance for FFT networks. *IEEE Transactions on Computers*, 43(7):849–854, 1994.

[153] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 579–586, July 2009.

[154] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.

[155] 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.

[156] 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*, pages 93–103, 2004.