SAND2022-10392C

# Integrating process, control-flow, and data resiliency layers using a hybrid Fenix/Kokkos approach

1st Matthew Whitlock
*Sandia National Laboratories*
*Livermore, CA*
*Georgia Institute of Technology*
*Atlanta, GA*
mwhitlo@sandia.gov

2nd Nicolas Morales
*Sandia National Laboratories*
*Livermore, CA*
nmmoral@sandia.gov

3rd George Bosilca
*University of Tennessee*
*Knoxville, TN*
bosilca@icl.utk.edu

4th Aurelien Bouteiller
*University of Tennessee*
*Knoxville, TN*
bouteill@icl.utk.edu

5th Bogdan Nicolae
*Argonne National Laboratory*
*Lemont, IL*
bnicolae@anl.gov

6th Keita Teranishi
*Sandia National Laboratories*
*Livermore, CA*
knteran@sandia.gov

7th Elisabeth Giem
*University of California*
*Riverside, CA*
*Sandia National Laboratories*
*Livermore, CA*
eagiem@sandia.gov

8th Vivek Sarkar
*Georgia Institute of Technology*
*Atlanta, GA*
vsarkar@gatech.edu

***Abstract*—Integrating recent advancements in resilient algorithms and techniques into existing codes is a singular challenge in fault tolerance – in part due to the underlying complexity of implementing resilience in the first place, but also due to the difficulty introduced when integrating the functionality of a standalone new strategy with the preexisting resilience layers of an application. We propose that the answer is not to build integrated solutions for users, but runtimes designed to integrate into a larger comprehensive resilience system and thereby enable the necessary jump to multi-layered recovery. Our work designs, implements, and verifies one such comprehensive system of runtimes. Utilizing Fenix, a process resilience tool with integration into preexisting resilience systems as a design priority, we update Kokkos Resilience and the use pattern of VeloC to support application-level integration of resilience runtimes. Our work shows that designing integrable systems rather than integrated systems allows for user-designed optimization and upgrading of resilience techniques while maintaining the simplicity and performance of all-in-one resilience solutions. More application-specific choice in resilience strategies allows for better long-term flexibility, performance, and — importantly — simplicity.**

***Index Terms*—Fault Tolerance, Resilience, Checkpointing, MPI-ULFM, Kokkos, Fenix, HPC.**

## I. Introduction

High Performance Computing is a critical part of research and many industries, with demand for compute resources continually growing. Longer running applications and larger, more complex clusters lead to higher rates of application failures. These failures cause significant losses in time and energy: an analysis of the Blue Waters system at the University of Illinois found that $\sim 9\%$ of the system's total production node hours were lost to system-software-induced failures alone, and node failures happened every 4.2 hours [1]. Oftentimes, hardware strategies to lower the ever-growing power consumption of systems trade reliability for power [2], [3]. Therefore, software resilience strategies are important for application robustness without sacrificing power efficiency gains. In this work we concern ourselves with the two primary layers of resilience (data and process recovery), and control-flow recovery - an often-overlooked third layer. Figure 1 displays these layers and the sample strategies we will be using, which will be discussed in Section V.

The most common data-level strategy is Checkpoint/Restart (C/R), in which applications write their data to a resilient storage space (typically disk). Process-level resilience is a newer concept for HPC applications, which primarily use MPI. The process level focuses on minimizing the disruption to local data and the cost of MPI relaunch by recovering without killing surviving processes after the failure of one process. Finally, the control-flow layer manages what and when to checkpoint/recover and how to resume execution after a failure.

Many prior studies have shown that each of these layers are necessary for comprehensive, performant resilience runtimes [4]–[6]. This is a problem when viewed in conjunction with the current state of resilience runtimes, which often either assume they are running alone, without any additional layers, or leave it to the application to manage potential complex integration of resilience layers. Runtimes which integrate internally with other layers directly suffer from a lack of versatility — as new hardware, data-recovery strategies, or parallelism runtimes emerge these pre-integrated resilience libraries are either rewritten to integrate with new runtimes or are deprecated.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE- NA0003525.

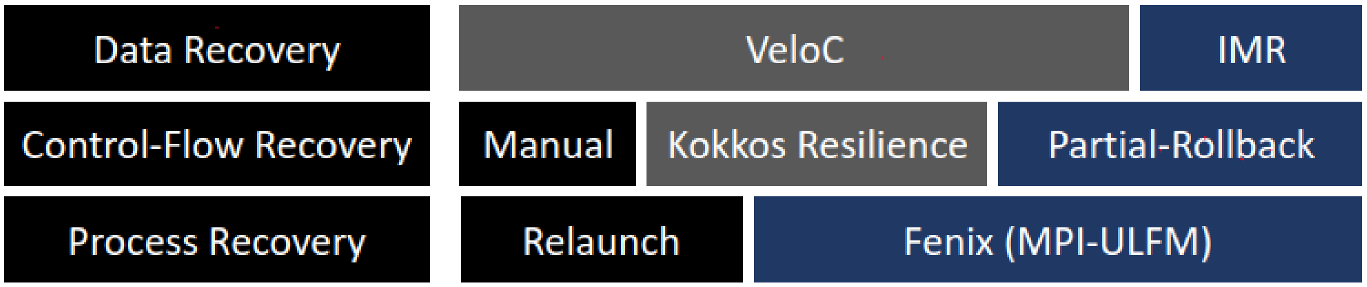


Fig. 1. Visualization of the recovery layers and the placement of resilience frameworks within them.

Realizing the full benefits of resilience at an application and systemic level relies on developing new algorithms and strategies for the various layers, and reaping the full benefits of these methods requires tailoring their integration per-application. Delegating the current complexity of this to users risks at best under-performing resilience and at worst resilience which fails to recover consistently or subtly gives incorrect results. The only way to enable comprehensive runtimes which use algorithms that are most performant and most hardware/software specific is by designing runtimes with integration into a larger, application-specific resilience strategy as a core design principle.

Our work with Fenix [7] as a process-layer resilience runtime designed specifically for simplifying integration into larger application-specific resilience systems demonstrates simplifications in development that do not sacrifice performance. We present a comprehensive resilience system runtime which integrates process resilience, control-flow resilience, and data resilience. Out implementation uses Fenix for the process resilience component, Kokkos Resilience [8] for control-flow resilience component, and VeloC [9] for data resilience. The result of this is a highly performant, comprehensive resilience environment consisting of runtimes that offer support for a wide array of state-of-the-art resilience strategies. More than performant, our work is simple both to maintain and add to applications; we show that implementing resilience with this system is minimally code-invasive, and updates to customize for specific application or platform features is simplified by exposing the integration of these runtimes to the application. These properties make for a comprehensive resilience system that is simple, performant, and flexible.

The rest of the paper is organized as follows. Section III discusses the runtimes that make up our system, and their related works. Section IV explains the protocols for our comprehensive resilience system. Section V details the work done to support our algorithm and the practical details of using in an application. Results on the code complexity and performance of our combined runtimes resilience are presented in Section VI. Finally, we offer the lessons learned in this process and a roadmap for a resiliency community focused on comprehensive and reliable resilience in Section VII.

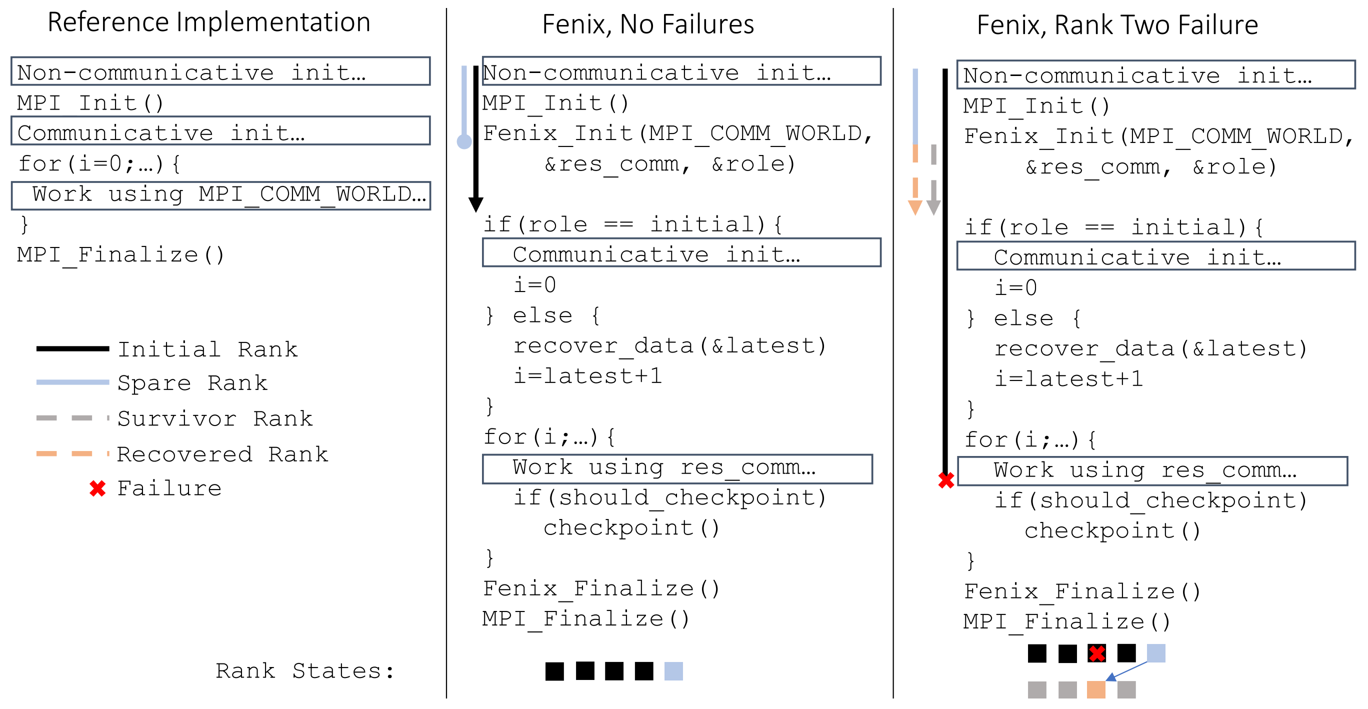
## II. Related works

Work regarding data resilience is prolific and has been for some time now. There are works such as VeloC [9], DMTCP [10], CPPC [11], and SCR [12] which focus on simplifying or automating performant checkpointing. Further, there is much investigation into improving support for checkpointing across heterogeneous devices: including FTI [13] and CRUM [14]. These libraries each present a unique set of pros and cons that make the ideal choice highly dependent on application and environment details.

The control-flow resilience layer is a newer focus of research. Kokkos Resilience [8], [15] and Resilient HCLIB [16] both utilize existing parallel control libraries to manage control-flow during recovery, which benefits from preexisting knowledge of the application's default control-flow. The two implement resilience methods in parallel-region and asynchronous-many-task runtimes, respectively. In addition, the CPPC data resilience system discussed above automates control-flow recovery without relying on a specific method of parallelism by using compiler-based tooling.

There are also several ongoing process resilience research projects. User Level Fault Mitigation (ULFM) [17] proposes a set of additional MPI functions to enable dynamic failure recovery and management at the cost of being somewhat complex. MPI-Reinit [18] is another MPI-based work that focuses on a simplified restart process at the expense of being less flexible in how recovery is handled. Fenix [7] attempts to bridge the gap between the two by using ULFM and automating application rollback and communicator recovery but allowing for users to selectively tune portions of code with lower-level controls. There are also a sampling of non-MPI options – e.g. FMI [19] and ACR [5] – which can provide enhanced and highly-automated process resilience at the cost of requiring application rewrites to a new message-passing library.

There are many works which integrate multiple runtimes and resilience layers and demonstrate the performance benefits of doing so. Automatic Checkpoint/Recovery (ACR) [5] builds a runtime which handles process, control-flow, and data resilience altogether. Checkpoint-Restart and Automatic Fault Tolerance (CRAFT) combines their control-flow resilience with ULFM process resilience and either their own C/R system or the SCR runtime, and FTI similarly has recent work in



```
Non-communicative init…
MPI Init()
Communicative init…
for(i=0;…){
 Work using MPI_COMM_WORLD…
}
MPI_Finalize()
```

```
Non-communicative init…
MPI_Init()
Fenix_Init(MPI_COMM_WORLD,
    &res_comm, &role)

if(role == initial){
  Communicative init…
  i=0
} else {
  recover_data(&latest)
  i=latest+1
}
for(i;…){
  Work using res_comm…
  if(should_checkpoint)
    checkpoint()
}
Fenix_Finalize()
MPI_Finalize()
```

Fig. 2. Visualization of the Fenix algorithm and typical control flow.

integrating with MPI-Reinit [20]. The Habanero C-Library (HCLIB) [6] offers an extension that integrates with Fenix for process resilience and VeloC for C/R. Prior work on integrating ULFM with the Compiler for Portable Checkpointing (CPPC) [21] uses a compiler-based instrumentation approach to automate recovery using ULFM and the CPPC library for data resilience. Even the Kokkos Resilience work we utilize here integrates a custom control-flow recovery with VeloC for data recovery.

These works form an important corpus on gauging the resilience patterns of integrated layers, but lack the future-proofing of simplifying the integration in the first place. They represent the pre-integration that is inherently less future-proofed and flexible than a resilience environment focused on simplified application-level integration. Our work is distinct from these in that it is designed to be used as a larger resilience runtime scheme while maintaining the simplicity and performance of resilience that prior work aims for. This enables performance that continues to improve as new runtimes, strategies, and technologies emerge; and tailoring resilience precisely to applications and hardware. We enable greater heterogeneity support by enabling the MPI+X model directly, simplify implementing high-performing resilience in both applications that have and lack preexisting resilience, and leave enough control to developers to enable low-level optimizations when desired.

## III. Background

Clearly, resilience in high performance computing is a well-established field with a large number of preexisting libraries for process recovery, control-flow management, and data recovery. We have chosen representative runtimes which are high performing, novel, and contemporary.

Our chosen data-level runtime is VeloC, a recovery runtime whose use involves a data C/R server application launched on each node. VeloC has users register checkpoint regions, then uses the co-located server thread to asynchronously manage migrating data to available resilient storage spaces based on dynamic information on space and performance of the spaces. VeloC can manage distributed parallelism for the user, but this functionality does not accommodate changes in the distributed thread-space. This makes integration with even simpler process-layer runtimes more complex for the user.

The control-flow runtime we use is Kokkos Resilience, a project built on top of Kokkos [22], a parallel C++ programming model designed to simplify performance portability for parallel applications. Kokkos Resilience uses Kokkos's model of data storage and functor- and lambda-based parallelism to automatically detect the data to be checkpointed and safe locations to checkpoint/recover. Kokkos Resilience can automatically detect data being used deep in nested function calls, so users cover large recovery regions with just top-level code adjustments. Kokkos Resilience is designed to pass its knowledge to various C/R backends, simplifying pre-integration with other runtimes but not necessarily enabling simplified user-space integration.

Our process-level runtime Fenix is built to simplify comprehensive resilience using ULFM [17]. The ULFM specification adds a small number of versatile functions which allow for reporting, recovering, and propagating failures in MPI applications. Failures can be reported at any MPI function call, which makes integration with control-flow layer runtimes which assume fail-restart semantics difficult. Further, recovery after failures principally hinges on shrinking communicators to exclude failed ranks. This complicates integrating with typical C/R libraries like VeloC, as it does not preserve rank IDs and changes the required data distribution.

## IV. Protocol of our Integrated Resilience System

To better understand of the design of our framework, we first discuss the typical control-flow of process resilience with Fenix. The purpose of Fenix is to simplify MPI process resilience by providing two primary benefits: (1) maintaining a resilient communicator which appears to have a consistent process pool even after process failures; and (2) reducing the number of failure states by making a single control-flow exit point for failures. The façade of a consistent resilient communicator is managed by holding some ranks out of the resilient communicator until after a failure, at which point they replace failed ranks. From an application perspective, these spare ranks block at the Fenix initialization call. A single control-flow exit is formed by attaching an error handler to the resilient communicator which performs a long-jump back to the Fenix initialization location whenever failures are detected.

Figure 2 visualizes the typical layout and failure-response of a Fenix-enabled application, and aligns closely with the benchmark application we use (discussed in Section VI). Typically, users initialize Fenix shortly after MPI and before performing any rank-dependent setup or communication. Principally, this initialization takes an input communicator (typically `MPI_COMM_WORLD`) and returns a new ”resilient communicator” which excludes the user-specified number of spare ranks from the input communicator. Fenix specification does not require that it is initialized before any MPI communications, but care must be taken to account for spare ranks that will not progress past the initialization call. After initialization, users proceed as normal using the resilient communicator in place of the input communicator. Users eventually call the Fenix finalization function, typically immediately followed by finalizing MPI.

When failure on the resilient communicator is reported by ULFM, a Fenix recovery callback is called which handles several important actions. First, Fenix ensures that the failure information is propagated to all the other ranks, including the spares, thus the error is propagated not only in the resilient communicator but in the input communicator (possibly `MPI_COMM_WORLD`). Then, Fenix “repairs” the resilient communicator by replacing it with a communicator of the same size, but where all failed ranks have been replaced in-place with spare ranks. Finally, Fenix performs a long-jump back to the Fenix initialization call and runs any application callbacks before returning control to the application. At this point, it becomes important to monitor the “Fenix role” which is also returned by the initialization function; these take the form of the rank states shown in Figure 2 and give the application the ability to reason about the current state of a rank for C/R purposes.

With this process-layer protocol in mind, we can discuss the comprehensive resilience framework. Figure 3 gives an overview of how the three resilience layers interact. All data resilience is handled by VeloC, which is primarily used as a standalone tool without much feedback to and from the other layers. Kokkos Resilience manages the control-flow resilience, and as a middle layer has much more interaction with the others. As part of handling the control-flow changes necessary for failure recovery, it fully integrates VeloC internally and manages making all calls to VeloC for C/R. This means it must pass down any information VeloC needs from MPI - and in this case Fenix. Using Kokkos Resilience, applications simply wrap checkpoint regions (such as the inside of a for-loop) in a lambda passed to a checkpoint management function. This function automatically gathers metadata on the data regions to checkpoint, passes that information to VeloC, and handles checkpointing at user-configured intervals or recovering when capable/necessary.

Fenix handles detecting failures, repairing the communicator state, and reporting the failure after returning the application to its initialization. This reported information is used to inform Kokkos Resilience of failures, which requires replacing the old communicator, clearing and re-fetching its checkpoint metadata (as a checkpoint finished locally may not have finished globally), and managing recovery of data based on this information. It must also update cached information in itself and VeloC on the current rank ID, in the case that an application is capable of continuing execution with a shrunk communicator after running out of spare ranks.

## V. Implementation

Ideally, using these libraries would enable us to merge their runtimes into our protocol with only application-level code, and adding each layer into the algorithm would require little change to how we use the others. In fact, the process required changing the way we use VeloC and modifications to the internals of the Kokkos Resilience library.

We start with VeloC. The typical initialization call takes an MPI Communicator as input and does not include the functionality to replace this communicator. VeloC also allows a non-collective implementation, though this prevents it from automatically finding the best globally-available checkpoint. Using Fenix process recovery with VeloC requires using the non-collective mode of VeloC and manually performing a reduction operation to obtain a globally-best checkpoint. Fortunately, this changes only a few locations in a typical application.

The Kokkos Resilience interface does not provide the required functionality to operate simultaneously with both Fenix and VeloC. The VeloC backend in the library does not allow initializing VeloC in single mode, and contains state-based information which cannot be reset after a process failure. As part of this work, we have developed support for these issues into the Kokkos Resilience library. Our changes add

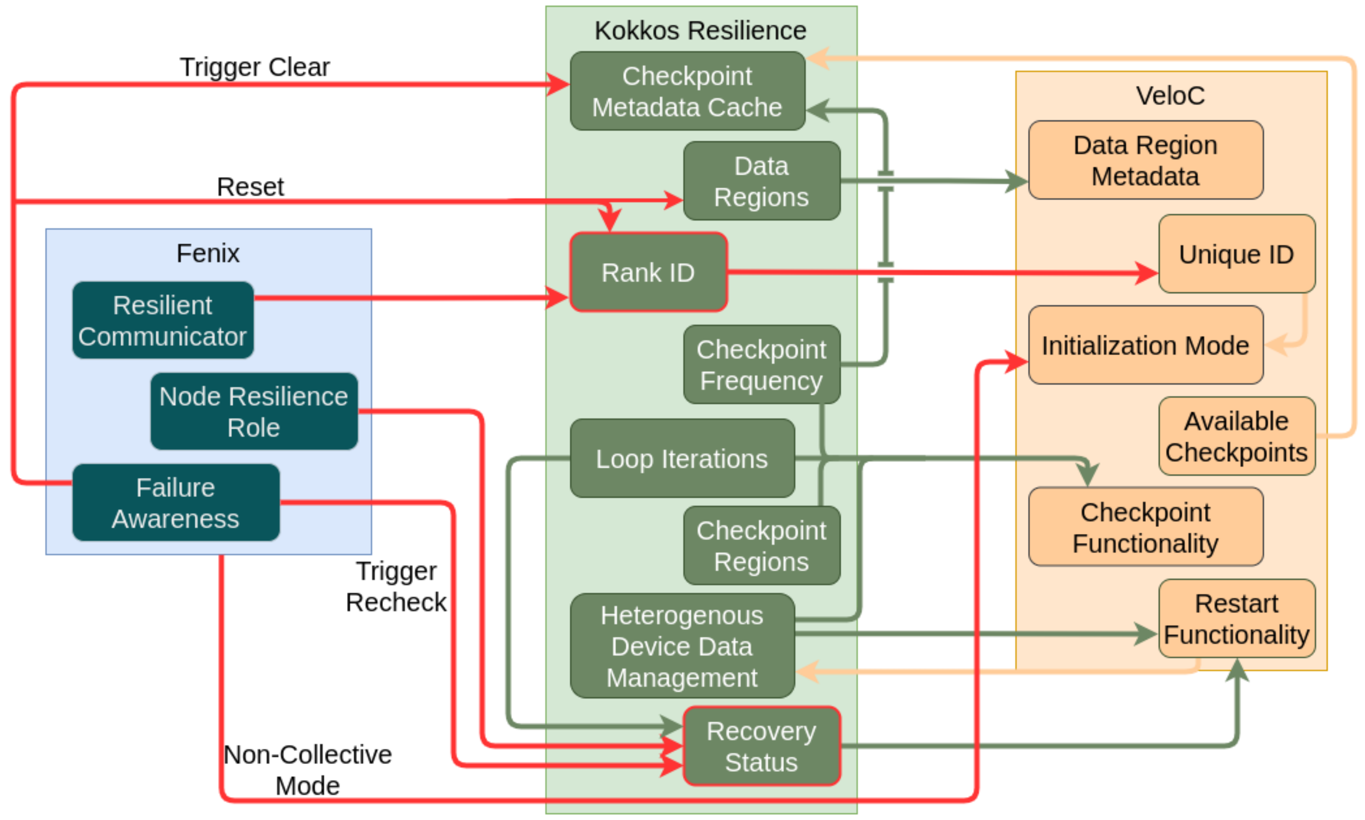


Fig. 3. Flowchart of our system of resilient runtimes. The heterogeneous support from Kokkos Resilience is not explored in this work. Red indicates portions implemented or modified to support our work.

an alternative new configuration option which launches VeloC in non-collective mode and handles performing the requisite communication to find the best available checkpoint. Further, we extend the Kokkos Resilience context reset function to also accept a new communicator and update its internal MPI information.

With these changes to support process-level resilience, our algorithm can be implemented in Kokkos-based application code with very little code modification and minimal knowledge of the inner details of the application. Figure 4 shows the typical use pattern.

### A. *Alternative implementations*

We have created implementations which explore other combinations of the methods for each layer presented in Figure 1. First as references we have configurations which use: VeloC alone; Kokkos Resilience without Fenix; and Fenix with VeloC but without Kokkos Resilience.

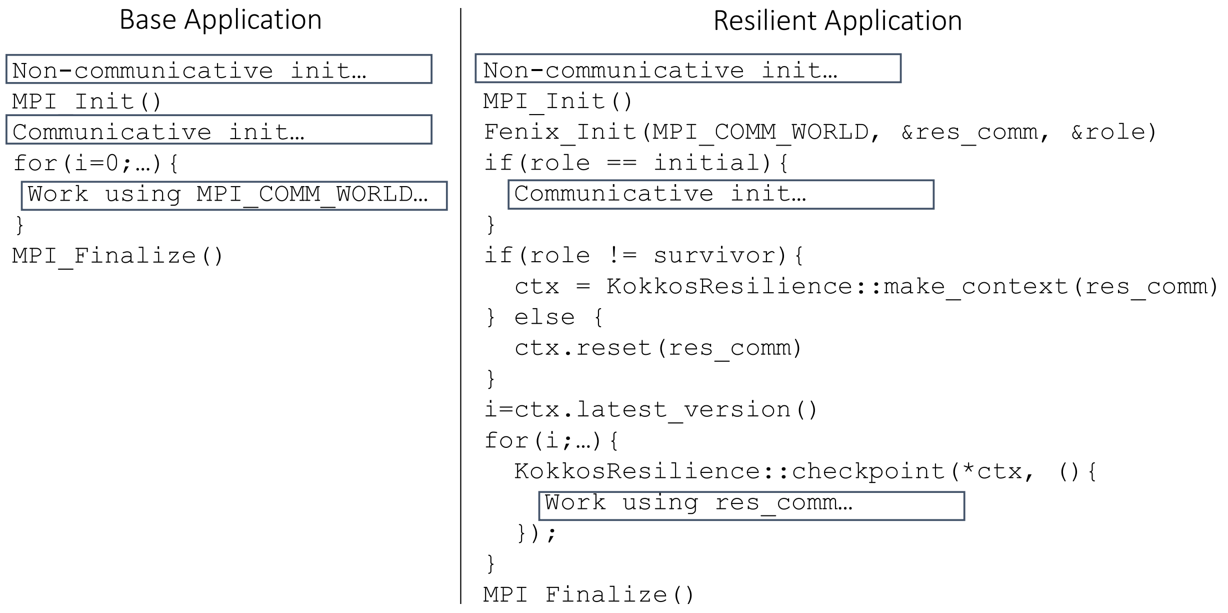
As demonstrations of the possible types of recovery enabled by process-level resiliency we have two more unique implementations. First, we utilize one of Fenix's built-in In Memory Redundancy (IMR) data resiliency policies. The IMR policies benefit from process-level resiliency by storing checkpoint data in the memory of other ranks, similar to buddy checkpointing approaches. In this work we use the *buddy rank* policy, in which ranks form pairs and store each other's checkpointed data. Local copies of checkpoints are also kept, increasing memory use in exchange for quick, local recovery on surviving ranks.

Second, we extend VeloC's Heatdis benchmark to benefit from the progress surviving ranks have made since the last checkpoint. Since our process resiliency layer leaves survivor ranks rather than destroying them all and relaunching, work done since the most recent checkpoint on the survivors may still be sitting in the application's data copy. Some applications, such as many iterative solvers which converge on a solution, can tolerate the error induced by the partial consistency of having failed ranks use an older iteration's data. We present a demonstration of this partial rollback method which skips checkpoint recovery on surviving ranks but computes until the difference between iterations is below a threshold. We use Fenix and Kokkos Resilience to manage the resilience of this extension, which required further modifying Kokkos Resilience to enable restoring at just one rank with VeloC.

## VI. Empirical Evaluations

### A. *Applications*

We use two sample applications to demonstrate our work. First, the VeloC heat distribution benchmark (Heatdis), modified to use Kokkos for parallelism control. The application is a 2D stencil that runs for a static number of iterations


Base Application
Resilient Application

```
Non-communicative init…
MPI_Init()
Communicative init…
for(i=0;…){
  Work using MPI_COMM_WORLD…
}
MPI_Finalize()
```

```
Non-communicative init…
MPI_Init()
Fenix_Init(MPI_COMM_WORLD, &res_comm, &role)
if(role == initial){
  Communicative init…
}
if(role != survivor){
  ctx = KokkosResilience::make_context(res_comm)
} else {
  ctx.reset(res_comm)
}
i=ctx.latest_version()
for(i;…){
  KokkosResilience::checkpoint(*ctx, (){
    Work using res_comm…
  });
}
MPI_Finalize()
```

Fig. 4. Demonstration of the typical steps to implement our system in an application.

and checkpoints by iteration count. To demonstrate a partial-consistency based strategy, we have further modified a version of Heatdis to run until data convergence. Heatdis has a configurable per-node application data size, which impacts the amount of processing as well as the amount of checkpointed data. All tests with Heatdis perform 6 checkpoints, which are each half the size of the application's data.

Second, we use the mini-app version of Sandia's Molecular Dynamics software, MiniMD [23]. MiniMD gives a more real-world sized example of implementing resilience, and is used to demonstrate the ease with which developers can use these combined strategies for performant resilience. We have gathered data on how our systems manage resilience with minimal development work; our goal is to explain the complexities in its communication and data movement patterns and examine how well these are handled.

### *B. Platform*

All tests performed for this work were conducted on a 100-node Cray XC40 system. Our tests were run on 2-socket Intel Haswell CPU nodes with 32 cores/node, and use disk-based checkpointing stores to the Lustre distributed file system.

### *C. Testing Details*

Since part of the recovery cost of relaunch-based resilience is the time required for shutdown and restart on all of the nodes, timing data cannot be exclusively gathered within applications. To supplement, we measure the time of the `mpirun` command using the `time` bash utility. Each tested application is run four times, twice with failure and twice without. The times are averaged in the reported overheads.

Failures are simulated through a rank exiting early, approximately $95\%$ of the way between two checkpoints. This ensures that the asynchronous VeloC checkpoints have completed prior to the failure, which is a more typical failure pattern for full-sized applications which use checkpointing.

VeloC is configured to use a filesystem folder mapped to local memory for scratch checkpoint storage, which means the synchronous portion of the VeloC checkpoint is just a memory copy of the application's data.

In Figure 5 we present the data gathered by checkpointing and recovering the Heatdis benchmark with varied data sizes and across a variable number of nodes (one rank per node, weak scaling data size). We separate the time spent in the application's local compute and time spent waiting on MPI function calls as "App compute" and "App MPI", respectively. The difference in in-app measurements and the bash-reported times are shown in Figure 5 as "Other", which covers data initialization, MPI job startup/teardown, and finalization time.

Similarly, in Figure 6 we present weak scaling performance data for the MiniMD application made resilient with our new framework. Here, we use MiniMDs existing profiling structures and report the application's execution time according to sections of compute type. "Force Compute" is a section of the application that is almost entirely compute-bound, with very little communication between ranks. "Neighboring" has more communication, but is still primarily local-compute-bound. However, "Communicator" represents the bulk of the communication in the app and is almost entirely communication-bound. As before, "Other" is the unaccounted-for time reported by the bash system and encompasses the same sources of overhead.

### *D. Performance*

As expected, Figure 5 shows no or negligible overhead is introduced by using Kokkos Resilience as a manager for VeloC checkpointing. This holds true when integrated with Fenix, as well. Of note, we see from Figure 5 that even

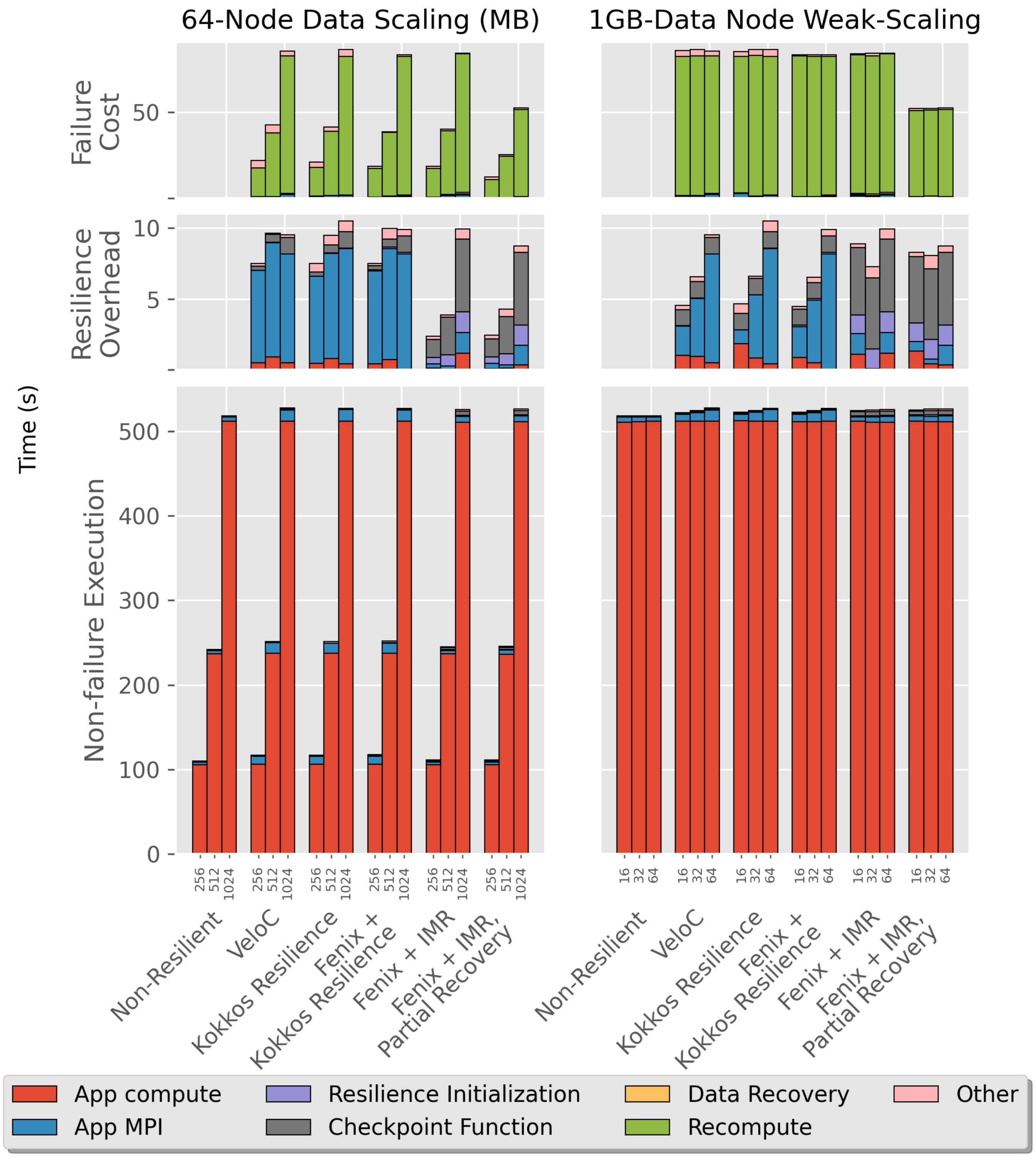


Fig. 5. The overhead and recovery costs for the resilience strategies implemented in the Heatdis benchmark, plotted against both the amount of application data and the degree of parallelism.

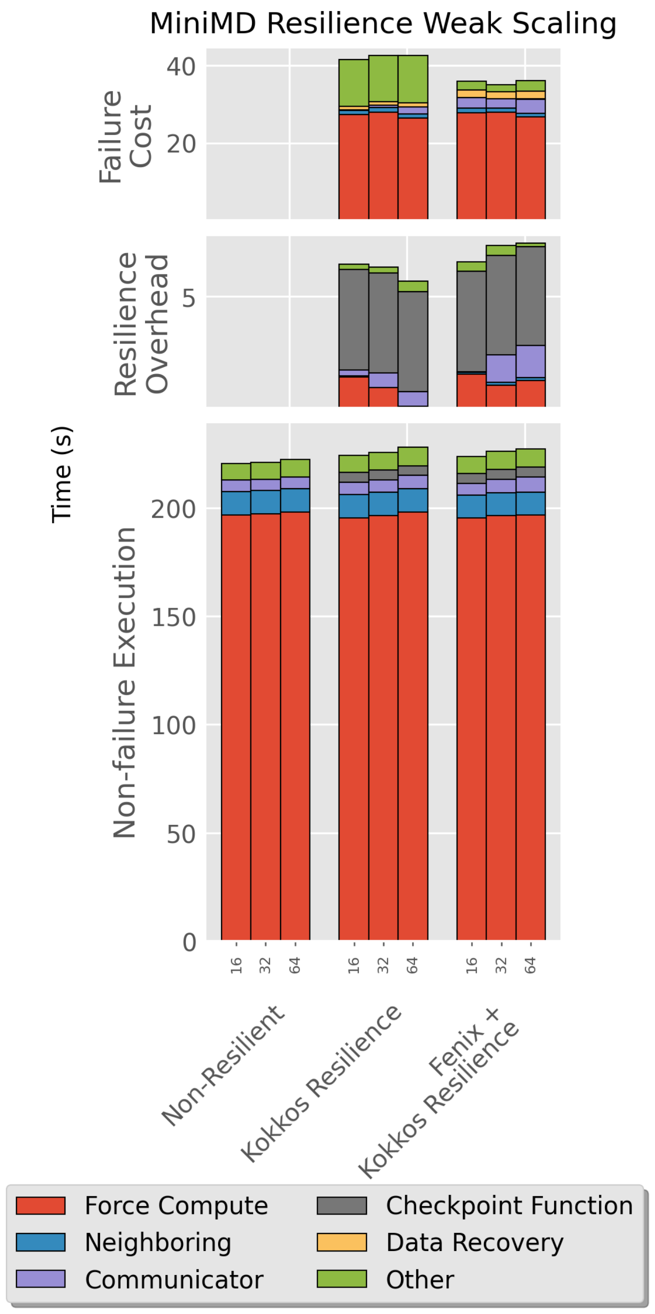


Fig. 6. The overhead and recovery costs for our resilience framework with MiniMD, plotted against the number of ranks executed on.

a naive implementation of Fenix improves performance of failure recovery with no or negligible additional overhead compared to a VeloC/Kokkos Resilience. The savings for using Fenix are primarily expressed through the "other" category, which points to the savings coming from avoiding the full tear-down and restart of all processes in the job.

*1) Checkpointing:* The total costs and the way they present vary between the two methods of resilient data storage. We see in Figure 5 that when using VeloC, the direct cost on calling the checkpoint function is quite low, just the amount of time it takes to make an in-memory copy of the data. Instead, the costs present primarily in slowing down the application by placing more of a burden on the network system. This congestion is more costly than IMR at low data sizes since the checkpointing is bottlenecked by the performance of the distributed filesystem – a large number of nodes are all at once writing data through a much smaller number of filesystem management nodes. During the time that the VeloC servers are congesting the network with their asynchronous writes to persistent storage, application MPI calls are delayed. However, we notice that the costs maintain very good scaling against the size of the data for these test cases. This is because while the maximum throughput of the filesystem extends the time required to write the checkpoints, it also places an upper bound on the amount of congestion that can be generated by the process of moving checkpoints to persistent storage.

On the other hand, IMR based checkpointing is much more direct in the scaling and presentation of its costs. We see that the checkpoint function costs scale directly with the size of the checkpoint, and that it scales worse against data size than VeloC-based checkpointing. However, at lower values it significantly outperforms disk-based storage since it better utilizes the available network bandwidth by distributing the data destinations rather than bottlenecking on the number of filesystem management nodes. We also see better scaling against the number of ranks being run across, since each rank adds both a producer and a consumer of the checkpoint data. This allows us to scale very well until hitting the overall network bandwidth limits, and effective network interconnects can help this scaling continue working on higher node counts.

The nature of this comparison depends heavily on the application's compute and communication patterns. Since communication is delayed, if it overlaps compute efficiently or is the primary cost for portions of the application we will see the impact of asynchronous writes to the filesystem vary significantly. This can be seen in Figure 6 by how checkpointing impacts the cost of the different portions of the application. The mostly compute-bound "Force Compute" and "Neighboring" portions of the application have relatively little overhead compared to the much more significantly relative cost to the communication-bound "Communicator" sections.

In fact, we can see that as node counts rise, the overhead of asynchronous checkpointing that presents in the force computing section of Figure 6 lowers. This is due to the higher performance variability across larger node counts, which naturally delays MPI communications based on the last rank to enter the MPI portion of the code and can hide some amount of the increased latency introduced by the asynchronous checkpoint process. As an artifact of performance variability (in compute and in network communications), a type of system noise in essence, this effect is itself highly subject to noise.

*2) Recovery:* Well optimized recovery behavior in both VeloC and IMR checkpointing limits the data-recovering cost of a restart significantly. Both methods only require network communication to restore the state of the failed rank(s); other ranks are able to restore using locally-available checkpoint

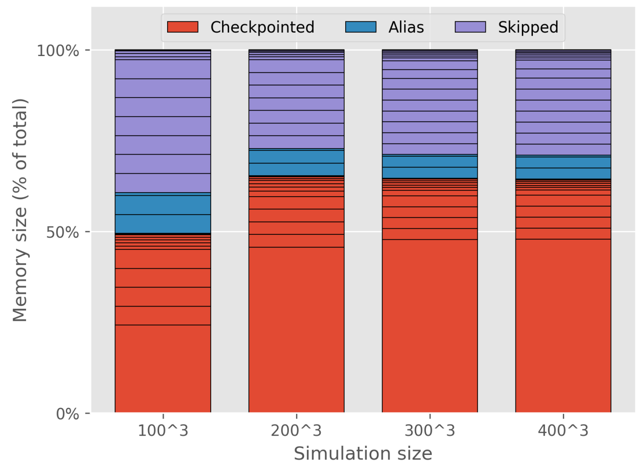


Fig. 7. Statistics on the relative sizes of the data regions of MiniMD and how they are checkpointed or ignored.

files. This means the cost of data recovery is similar between the two, and in these tests scales directly with the size of the data being recovered. Given the asymmetrical recovery cost, the ability of quickly-recovering ranks to make progress past the slowly-recovering ranks will determine the effective cost of data recovery.

Another cost of recovery is in resetting the application process' states. For the non-Fenix implementations, resetting the states requires relaunching MPI entirely and redoing all initialization (commonly including reading configuration files, allocating large memory regions, etc.). We can see that the higher initialization cost associated with MiniMD leads to greater savings in Figure 6 by the significantly smaller "Other" category in the failure costs for the Fenix-enabled resilience.

However, the bulk of the cost of recovery is in recomputing the data lost since the last checkpoint. In Figure 5 this recompute cost is explicitly separated; in Figure 6 it can be seen as the extra time spent in the application's three compute phases in the failure costs. Here we see much larger benefits to the use of Fenix – it enables surviving ranks to keep their in-progress data. For applications which can handle an inconsistent state across ranks (either because it is resilient to data errors or because one rank using older data simply does not introduce any error), this means the amount of recomputation may be lowered significantly. In our example of the heat distribution application iteratively lowering the error, we see a nearly 2x speedup of recovery from just keeping the in-progress data on surviving ranks — a very easy and very effective optimization when it is applicable.

## *E. Complexity of Use*

After converting the MiniMD mini application to support Kokkos Resilience and Fenix, we have gathered some quantitative figures in an attempt to convey the ease of use of this resilience system and how effective it is with what amount and type of code we have written. To begin, Figure 7 shows how memory regions are broken up within the mini app and which ones are checkpointed. Alias views are views which the user has manually specified to contain the same data as their alias, and so should not be checkpointed. Skipped views are duplicate copies of a view which are copied into the checkpoint region.

We can first note that a single view contains the majority of the data for the application, with only a handful making up nearly all of the checkpointed data. This means that while there are a total of 39 views checkpointed, manually inspecting only a small sample of 67 is sufficient to minimize checkpoint size to a large degree. We have also observed that three views are marked as aliases of another, accommodating a temporary swap space for the application. Thus, developers can simply list the two view labels as being aliases for each other to indicate these swap-space views should not be checkpointed.

Further, we can see the large memory size of the 19 skipped views. These views are already checkpointed but stored a second time in additional objects that are copied into the checkpoint lambda by the compiler; they represent views which are used across multiple sources and which developers would need to carefully register once and only once with a checkpoint library like VeloC. These are automatically detected by Kokkos Resilience so as not to be checkpointed multiple times.

In addition to the 61 view objects in the application, over the 20+ source files 15 of them collectively contain over 148 locations with MPI code. With a typical ULFM error handling approach, each of these would need to be adapted to support error handling, and the number of error states is very large. Using Fenix we can simply swap references to `MPI_COMM_WORLD` to the resilient communicator used by Fenix and then add in fewer than 20 lines of simple code to a single file. Adding calls to Kokkos Resilience is similarly simple.

## *F. Lessons Learned*

We have discussed the implementation and performance of our integrated resilience runtime framework with specific software components and configurations. Design and implementation of the whole framework involved a significant amount of work to modify the individual components and their couplings. We now discuss some key lessons learned towards scalable and reusable resilience runtime software for future HPC systems.

*1) Flexibility is key:* Optimal resilience strategies depend heavily on the application and environment details. Even within a single application, variations in node count and application data size quickly change which of our checkpointing methods performs best. Supporting the flexibility required by application developers ought to be a major priority of resilience libraries.

*2) Automate, but design for exceptions:* A common thread between Kokkos Resilience and Fenix is that the standard use-case is highly automated. Implementing either with the expected patterns takes little effort and little code knowledge, and it achieves reasonably good performance from the start.

However, flexibility remains key – as demonstrated by Figure 7, allowing users to investigate and manually manage resilience piecemeal can greatly lower the total amount of effort by letting developers target their efforts on high-impact regions of code.

*3) Layering is ideal:* The default Heatdis application is not especially suited to online process recovery – it requires synchronous recovery, performs global operations often, and demands all data be rolled back to maintain a consistent data state between ranks. Even so, it benefits from using a process resilience layer: this demonstrates the widespread utility in simplifying the use of every layer of recovery.

## VII. Conclusion

With small changes to the structure and type of APIs resiliency programs provide, user-level code modifications can be significantly lowered for complex applications while still gaining nearly the same levels of performance and resilience as resilience hand-tuned with lower level tools. Our approach of sharing information between the resilience layers without requiring tightly coupled libraries in a single tool means users can tailor the specifics of the executions to their individual applications and environments. Enabling resilience that combines simplicity and flexibility through communication between resiliency layers can be a larger focus of resilient libraries going forward as the number of libraries and specific functionalities expand. Otherwise we contribute to a growing burden on developers who are already reticent to include more than the most basic of resilience in their applications. In short, resilience — and specifically layering resilience — is hard, and simplifying it is both necessary and feasible through communication between the layers.

### A. Future Work

Further integration of Fenix and Kokkos Resilience in the form of a data-resiliency backend is a great goal. Further, adding a new backend tier to Kokkos Resilience for process resiliency libraries would enable even more simplification and open the door for more process resilience strategies. This would remove the need for two resilience initialization steps, and further lower the amount of control-flow modifications needed for implementing the combination of Fenix and Kokkos Resilience. So long as simple integration with other layers from the outside continues to be maintained, closer integrations can serve as a convenience without forcing users into specific resilience combinations. Testing and designing our system for heterogeneous systems is another clear step which could simplify managing resilience.

More significantly, improvements to integrations with the process recovery layer can enable more complex recovery patterns without burdening the user. This includes techniques like shrinking and growing the total number of ranks dynamically throughout execution and migrating processes for post-failure load balancing. These strategies exist in some of the related works discussed, but enabling them without requiring developers to switch to new communication libraries would be a great step in both simplifying and improving flexibility.

## References

[1] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 610–621.

[2] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Z. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: A data-driven literature analysis," *ACM Comput. Surv.*, vol. 53, no. 2, mar 2020. [Online]. Available: https://doi.org/10.1145/3372390

[3] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge, and D. J. Kerbyson, "Investigating the interplay between energy efficiency and resilience in high performance computing," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 786–796.

[4] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance," 2017.

[5] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[6] S. R. Paul, A. Hayashi, M. Whitlock, S. Bak, K. Teranishi, J. Mayo, M. Grossman, and V. Sarkar, "Integrating inter-node communication with a resilient asynchronous many-task runtime system," in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 41–51.

[7] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar, "Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques," in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, 2016, pp. 346–355.

[8] N. Morales, K. Teranishi, B. Nicolae, C. Trott, and F. Cappello, "Towards high performance resilience using performance portable abstractions," in *Euro-Par 2021*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), L. Sousa, N. Roma, and P. Tomás, Eds. Germany: Springer, 2021, pp. 451–465.

[9] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 911–920.

[10] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[11] G. Rodríguez, M. Martín, P. González, J. Touriño, and R. Doallo, "Cppc: A compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 749–766, 04 2010.

[12] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[13] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal, "Checkpoint restart support for heterogeneous hpc applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 242–251. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CCGrid49817.2020.00-69

[14] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "Crum: Checkpoint-restart support for cuda's unified memory," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 302–313.

[15] Sandia National Labs, "Kokkos Resilience," 2022. [Online]. Available: https://github.com/kokkos/kokkos-resilience

[16] S. R. Paul, A. Hayashi, N. Slattengren, H. Kolla, M. Whitlock, S. Bak, K. Teranishi, J. Mayo, and V. Sarkar, "Enabling resilience in asynchronous many-task programming models," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed. Cham: Springer International Publishing, 2019, pp. 346–360.

[17] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, vol. 27, pp. 244 – 254, 2013-01 2013.

[18] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance," 2021.

[19] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. d. Supinski, N. Maruyama, and S. Matsuoka, "Fmi: Fault tolerant messaging interface for fast and transparent recovery," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1225–1234.

[20] K. Parasyris, G. Georgakoudis, L. Bautista-Gomez, and I. Laguna, "Co-designing multi-level checkpoint restart for mpi applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 103–112.

[21] N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín, "Local rollback for resilient mpi applications with application-level checkpointing and message logging," *Future Generation Computer Systems*, vol. 91, pp. 450–464, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18303443

[22] C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.

[23] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.